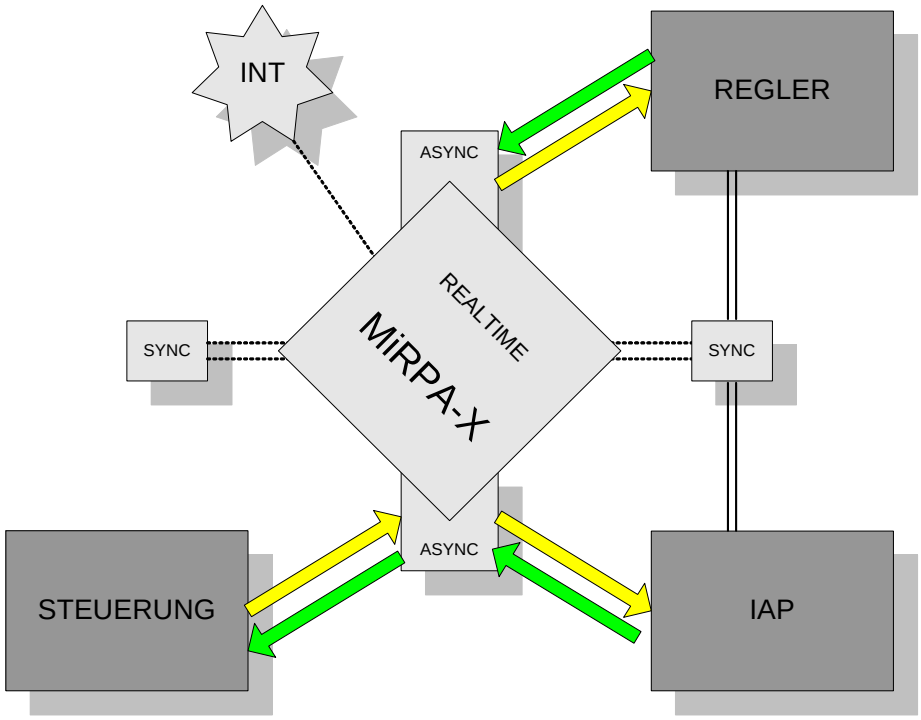
# Kommunikations-Infrastruktur für hochdynamische Parallelroboter

**Nnamdi Kohn**




**Berichte aus dem Institut
für Elektrische Messtechnik und
Grundlagen der Elektrotechnik**

Technische Universität Braunschweig
Prof. Dr.-Ing. J.-Uwe Varchmin

# Kommunikations-Infrastruktur für hochdynamische Parallelroboter

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung der Würde

eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation


von: Dipl.-Ing. Peter Nnamdi Kohn

aus (Geburtsort): Hamburg



eingereicht am: 4. Juli 2007

mündliche Prüfung am: 19. Oktober 2007


Referenten:

| | |
|---|---|
| Vorsitzender | Prof. Dr.-Ing. Walter Schumacher |
| 1. Berichter | Prof. Dr.-Ing. Jörn-Uwe Varchmin |
| 2. Berichter | Prof. Dr.-Ing. Harald Michalik |

2007

## Danksagung

Die Inhalte der hier vorliegenden Arbeit stützen sich auf meine Projektarbeit im Teilprojekt B1 „Steuerungs- und Kommunikationsarchitekturen“ des Sonderforschungsbereichs 562 „Robotersysteme für Handhabung und Montage“, das ich während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Elektrische Messtechnik und Grundlagen der Elektrotechnik (emg) der TU Braunschweig bearbeitete.

Das besonders angenehme Arbeitsklima am Institut und innerhalb aller Projekte, an denen ich beteiligt sein konnte, verdanke ich vor allem Professor Jörn-Uwe Varchmin, der mir stets mit Ideen, Rat und Tat zur Seite stand. Meinen Kollegen aus emg-alpha sei gedankt für ihren Beitrag zum durchweg freundschaftlichen und lockeren Miteinander, das sich gelegentlich auch mal auf Chefs Terrasse abspielte. Für den fachlichen und stets konstruktiven Beistand in unserer Arbeitsgruppe gegen Ende meiner Projektphase, danke ich besonders Professor Harald Michalik.

Dankbar bin ich für die kurzweilige Zeit, die ich mit all den Kollegen der unterschiedlichen Fachgebiete verbringen konnte, um Begehungen, Kolloquien, Messe-Auftritte, Vorträge und Veröffentlichungen, teilweise bis in die frühen Morgenstunden vorzubereiten. Dabei möchte ich besonders den Herren Stephan Algermissen, Matthias Bruhn, Karsten Diethers, Michael Kolbus, Jochen Maaß, Thomas Reisinger, Christoph Stachera, Jens Steiner und Frau Ulrike Thomas erwähnen. Sie haben permanent Ideen und Anmerkungen zugunsten der Anwendung und Erprobung der in dieser Arbeit vorgestellten Konzepte einfließen lassen. Für die stets humorvolle und freundschaftliche Schulter-An-Schulter-Arbeit an unseren FireWire-Entwicklungen bedanke ich mich besonders bei meinem Freund Tobias Möglich. Meinem fachlichen Vorgänger, Dr. Guido Beckmann, danke ich für seine detaillierte Vorarbeit und hilfreiche Tipps besonders zu Beginn meiner Tätigkeit. Meinem Nachfolger, Yannick Dadji, wünsche ich alles Gute bei der Realisierung weiterer Ideen in diesem immer komplexer werdenden Themengebiet.

Ich danke meiner Mutter, Irma Kohn, für ihre umfassende Unterstützung besonders während meines Studiums und die vielen persönlichen Impulse; auch meinen Brüdern Moses und Victor, auf die ich sehr stolz bin; meiner lieben Frau Hanna für Geduld, Motivation und Verständnis während der Phasen besonderer Belastung; und ganz besonders meinem Gott, der das Gelingen schenkt.

*Ich will dich unterweisen und dir den Weg zeigen, den du gehen sollst;*
*ich will dich mit meinen Augen leiten - Psalm 32,8*

*Und der Herr wird dich immerdar führen und dich sättigen in der Dürre*
*und dein Gebein stärken - Jesaja 58,11*

Braunschweig, im Juli 2007

# Inhaltsverzeichnis

# 1 Einleitung und Motivation

Die Anwendung von Robotersystemen zum Einsatz in der industriellen Handhabung und Montage stellt hohe Anforderungen an Wiederholgenauigkeit, Geschwindigkeit und Beschleunigung der Arbeitsplattform. Der Einsatz von Parallelstrukturen stellt dabei eine vielversprechende Alternative zur Verwendung konventioneller serieller Strukturen dar. Die zu verwendende Steuerungstechnik lässt sich allerdings nicht ohne weiteres aus der seriellen Anwendung ableiten, sondern erfordert die Berücksichtigung neuer und komplexer Fragestellungen aus den softwarebasierten Bereichen Roboterprogrammierung, Steuerungs-, Regelungs- und Kommunikationstechnik als auch aus den hardwarebasierten Bereichen Konstruktionstechnik und Sensorik.

Innerhalb des Sonderforschungsbereichs 562 „Robotersysteme für Handhabung und Montage“ werden methoden- und komponentenbezogene Grundlagen von Parallelrobotern erarbeitet, die an Demonstratoren in Betrieb genommen, getestet und angewendet werden. Für den Bereich der Kommunikationstechnik erfordert dies die Anwendung einer einheitlichen, flexiblen und modularen Schnittstelle. Sie soll Entwicklern aus den unterschiedlichen fachlichen Themengebieten in jeder Softwareschicht zuverlässige, schnelle und für die Anwendung passende Kommunikationsmechanismen zur Verfügung stellen und damit die Entwicklung eines solchen komplexen Robotersystems handhabbar machen.

Die vorliegende Arbeit knüpft an die Dissertation von Dr. Beckmann zur Spezifikation des Industrial Automation Protocol [21] an und stellt damit ihre Weiterführung mit dem Ziel der Realisierung einer geschlossenen, bedienbaren und für komplexe Robotersysteme nutzbaren Kommunikationsschnittstelle dar. Während die Dissertation von Dr. Beckmann sich mit der Konzeption eines für die industrielle Automation nutzbaren Kommunikationsprotokolls befasst, findet sich in der vorliegenden Arbeit die Konzeption und Realisierung einer übergeordneten Kommunikations-Infrastruktur. Sie schließt die erstmalige Implementierung der von Dr. Beckmann vorgeschlagenen Protokollspezifikation mit ein und nutzt diese als eine wesentliche Komponente innerhalb eines neuartigen Gesamtkonzeptes: eine echtzeitfähige Middleware zur vollständigen Entkoppelung von Anwendung (Robotersteuerung) und Kommunikation (über den IEEE1394-Standard).

## Motivation

Zur Steuerung serieller Roboter werden die einzelnen Bewegungsfreiheitsgrade unabhängig voneinander geregelt, da sich diese mittels offener kinematischer Ketten aufbauen lassen. Für den Betrieb eines solchen Roboters ermöglicht dieser Umstand die

Anwendung einer sogenannten „Motion-Pipeline“, bei der die Reihenfolge von Verfahrbewegungen einzelner Freiheitsgrade im Antriebskoordinatensystem für die Stabilität der Roboterstruktur unkritisch ist. Die Berechnungen der Antriebssollwerte sowie die zeitliche Applizierung hierfür können also für jeden Antriebsfreiheitsgrad separat erfolgen.

Serielle Roboter begünstigen aufgrund dieser kinematisch ungekoppelten Struktur die Anwendung eines dezentralen Steuerungsansatzes mit in den Komponenten untergebrachten Regelungsfunktionalitäten. Jede dezentrale Komponente steuert damit beispielsweise eine einzelne Bewegungsachse des Roboters. Mit diesem Ansatz können in der Regel auch aufwändige (übergeordnet koordinierte) Regelungsfunktionen realisiert werden, die nicht durch die verfügbare Rechenleistung beschränkt werden. Die Kommunikation von Sollwerten für die dezentralen Komponenten erfolgt dabei in einem Zeitraster, welches um Größenordnungen gröber ist als der Arbeitstakt der unterlagerten Regelungsfunktionen. Synchronisationsanforderungen und erforderliche Bandbreite sind dabei unkritisch, da die Übertragung der Positions- oder Geschwindigkeitsstützstellen für jede Koordinate unabhängig erfolgen kann.

Bei Parallelrobotern sind die Bewegungsalgorithmen aufgrund der Kopplung der Führungsketten über die Arbeitsplattform kinematisch und dynamisch abhängig voneinander. Diese Kopplungen lassen sich nur durch einen zentralen Steuerungsansatz handhaben, der einen direkten Quereinfluss zwischen den unterschiedlichen Antrieben ohne verzögernde Kommunikationsvorgänge auf Stellgrößenebene ermöglicht. Dieser zentrale Steuerungsansatz schränkt damit zu Bahn-Ungenauigkeiten führende Schleppfehler ein und ermöglicht es, innere Kräfte (Biegung, Stauchung und Torsion) in der Struktur zu handhaben und einzuschränken. In [100] sind die mathematischen Grundlagen für eine Modellbildung zur Analyse von Singularitäten beschrieben.

Während bei seriellen Robotern die Vorwärtstransformation (Berechnung der Position der Arbeitsplattform aus den momentanen Winkelkoordinaten der Antriebe, DKP[1]) analytisch gelöst werden kann, ist bei parallelen Strukturen im Allgemeinen ein iteratives Lösungsverfahren erforderlich. Das Gleichungssystem ist nichtlinear und weist in der Regel eine vollgekoppelte Struktur auf. Zusätzlich ist seine Lösung mathematisch nicht immer eindeutig, woraus unterschiedliche Lagen und Orientierungen der Arbeitsplattform im Raum resultieren. Dieser Mehrdeutigkeit kann durch das Verwenden struktur- und gelenkintegrierter Sensoren begegnet werden [131]. Sie ermöglichen eine eindeutige Lösung der Gleichungen und in speziellen Fällen sogar eine begrenzte Vereinfachung der Berechnungen.

Neben den beschriebenen Transformationsberechnungen fallen bei Parallelrobotern auch andere zu berechnende Funktionen rechenaufwändiger aus als bei seriellen Robotern. Während roboternahe Steuerungsfunktionen wie die Arbeitsraumüberwachung für serielle Roboter über das Abfragen und Überprüfen aktueller Antriebskoordinaten vollständig realisiert werden können, ist dies bei Parallelrobotern nicht ausreichend. Der Arbeitsraum ist hier nicht nur durch die Soft- oder Hardware-Endschalter der Gelenke begrenzt. Aufgrund der Besonderheiten bezüglich des von der Position und Orientierung des Endeffektors abhängigen Arbeitsraums [93] [95] und der Singulari-

[1] DKP = direct kinematic problem

täten innerhalb des Arbeitsraums [94] sind hier rechenzeitintensive Funktionalitäten der Überwachung notwendig.

Dem Vorteil eines zentralen Steuerungsansatzes, die zeiteffektive Berechnung kinematisch und dynamisch gekoppelter Gleichungen zu ermöglichen, steht die Forderung gegenüber, diese Berechnung in der Weise durchzuführen, dass durch den Rechenaufwand die eigentliche Dynamik der Verfahrbewegung, -geschwindigkeit und -beschleunigung des Roboters nicht beeinträchtigt wird. Außerdem kommt der Umstand hinzu, dass sämtliche errechneten Sollgrößen in dem Regelungs-Zeitraster (möglichst ohne zeitliches Jittern) an die entsprechenden Komponenten übertragen werden müssen.

Je schneller ein Arbeitszyklus des Roboters bei unverminderter Verfahrwiederholgenauigkeit abgeschlossen werden soll, desto größer muss die Frequenz gewählt werden, mit der die aktuelle Roboterpose erfasst und innerhalb der Steuerung neu berechnet wird. Diese Zyklusfrequenz wird maßgeblich durch die maximale Abtastfrequenz (gebildet aus notwendigem Zeitaufwand für Datenübertragung und -berechnung) nach oben hin begrenzt. Hinzu kommt, dass ein Jittern (temporäres Schwanken) der Abtastfrequenz zu einer temporären Verschlechterung der den Berechnungen zugrundeliegenden Modellgenauigkeit führt, was die inneren Kräfte innerhalb der Struktur verstärken kann.

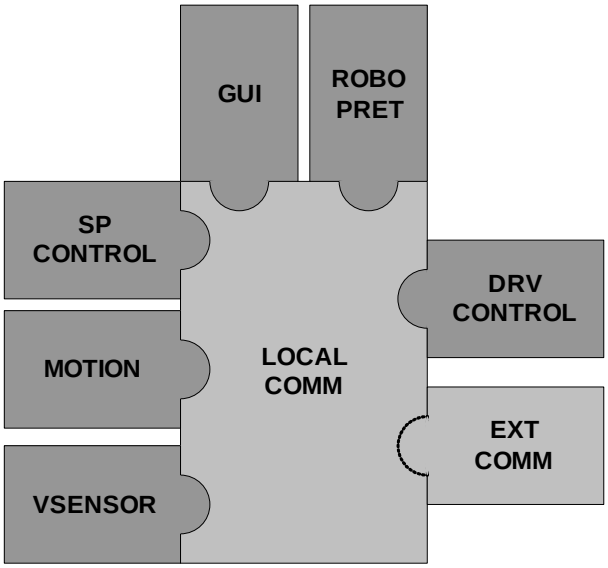
Auch bei der Anwendung kraftgeführter Montagesequenzen sind exakte Modelle besonders wichtig. Zur Realisierung einer definierten Strukturnachgiebigkeit der Arbeitsplattform ist eine exakte Kenntnis der aktuellen Roboterzustände unerlässlich, da bereits kleine Positionsungenauigkeiten zu großen Kräften bzw. Steifigkeiten innerhalb der Struktur führen können.

## Zielsetzung

Die oben dargestellten Anforderungen aus den Bereichen Steuerungs-, Regelungs- und Kommunikationstechnik erfordern die Anwendung einer effizienten, deterministischen, modularen Echtzeit-Kommunikations-Infrastruktur, deren Konzipierung und Entwicklung das Thema dieser Arbeit ist. Es wurde hierzu ein zentraler Systemansatz gewählt, um hybride Regelungsstrategien ermöglichen zu können, ohne dabei komplexe Regelungsfunktionalitäten zu dezentralisieren und damit strategieabhängige externe Kommunikationsbandbreite aufwenden zu müssen.

Abbildung 1.1 zeigt dazu die Softwaremodule der Zielanwendung auf dem zentralen Steuerungs-PC, die auch in den Forschungsarbeiten im Zusammenhang mit dem SFB 562 zum Einsatz kommen.

Ein Anwender definiert z.B. über ein grafisches Benutzerinterface (*GUI*) eine vollständige Montageaufgabe, die über einen Interpreter (*ROBOPRET*) in für die Steuerung verständliche Anweisungseinheiten umgewandelt werden. Die Steuerung (*SPCONTROL*) stellt den Rahmen für auf ein kartesisches Koordinatensystem basierende Funktionalitäten bereit, die zur Erzeugung einer Roboterbewegung notwendig sind




**GUI** - graphical user interface
**ROBOPRET** - robot program interpreter
**SPCONTROL** - skill-primitive control
**MOTION** - hybrid motion controller
**VSENSOR** - virtual sensing module
**DRVCONTROL** - subordinated drive control
**LOCALCOMM** - local communication
**EXTCOMM** - external communication

Abbildung 1.1: Softwarestruktur des Steuerungssystems im SFB 562

und koordiniert den Funktionsablauf auf verschiedenen Prioritätsebenen. In den Modulen *MOTION* und *VSENSOR* werden die kartesischen Berechnungen zur Bestimmung von Soll- und Istwerten der einzelnen Freiheitsgrade gekapselt. Die unterlagerte Antriebsregelung (*DRVCONTROL*) besteht aus Funktionalitäten, die eine geschlossene Regelung im Antriebskoordinatensystem durchführen. Diese Funktionalitäten sind fest mit einer gewählten Zyklusfrequenz verbunden. Das Modul *EXTCOMM* sorgt dafür, dass eine Verteilung der Soll- und Istwerte zu den Roboterkomponenten hin erfolgt. Alle zentral ablaufenden Kommunikationsvorgänge der Softwaremodule untereinander werden über das Modul *LOCALCOMM* koordiniert. In den Modulen *EXTCOMM* und *LOCALCOMM* sind die Funktionalitäten realisiert, die in dieser Arbeit konzipiert und entwickelt wurden.

Als Schnittstelle zwischen der Roboterprogrammierung und der Robotersteuerung werden im SFB 562 sogenannte Aktionsprimitive [91] verwendet, die jeweils einen mehrdimensionalen, sensorgeführten Bewegungsschritt des Roboters beschreiben und sich dabei auf ein kartesisches Koordinatensystem (Task Frame) beziehen. Zur Organisation und Verteilung steuerungsspezifischer Daten sind schnelle und namensbasierte Kommunikationsmechanismen erforderlich, die gleichzeitig die Modularität von Steuerungs- und zugehöriger Regelungsfunktionalitäten sicherstellen.

## Gliederung der Arbeit

In Kapitel 2 dieser Arbeit werden aktuelle Softwaretechnologien aus den Bereichen Betriebssysteme, Steuerungstechnik und Kommunikationstechnik vorgestellt. Hier werden grundlegende Begrifflichkeiten geklärt und über die Definition der Echtzeitfähigkeit spezifische Anforderungen für die in dieser Arbeit entwickelte Software aufgestellt.

Kapitel 3 beschreibt den aktuellen Stand der Technik im Bereich der seriellen Kommunikationssysteme. Mit der Vorstellung von Neuentwicklungen und ihren technischen Eigenschaften werden die Hardware-Grundlagen beschrieben, auf die zur Entwicklung einer Kommunikations-Infrastruktur für hochdynamische Robotersysteme zurückgegriffen werden konnte.

Kapitel 4 ist der Hauptteil dieser Arbeit und stellt die modulare Kommunikations-Infrastruktur vor, die im Rahmen der wissenschaftlichen Tätigkeit innerhalb des SFB 562 konzipiert und in Soft- und Hardware realisiert wurde.

Anhand des oben bereits erwähnten Anwendungsbeispiels aus dem Sonderforschungsbereich 562 wird im Kapitel 5 die Funktionalität der entwickelten Kommunikations-Infrastruktur erläutert und mittels Messungen bestätigt.

Eine Zusammenfassung und ein Ausblick auf zukünftige Arbeiten im Zusammenhang mit der vorgestellten Kommunikations-Infrastruktur sind Themen des Kapitels 6.

# 2 Softwaretechnologien

In diesem Kapitel werden allgemeine Softwaretechnologien erläutert, die im Zusammenhang mit der in dieser Arbeit vorgestellten Kommunikations-Infrastruktur stehen und Rahmenbedingungen für deren Entwicklung festlegen. Die Ausführungen sollen dazu dienen, den Stand der Technik für diesen Anwendungsbereich darzustellen.

Abbildung 2.1 zeigt ein Schichtenmodell, das die Aufteilung von Software- und Dienstklassen für allgemeine (auch verteilte) Systeme wiedergibt. Es wird hier zwischen Systemsoftware (Betriebssystem) und Middleware unterschieden, wobei letztere mögliche Kommunikations- und Steuerungsarchitekturen als Rundumlösung für Steuerungsaufgaben, Software-Schnittstellen der industriellen Steuerungstechnik für Kommunikationsaufgaben und allgemeinen Middlewarelösungen für spezielle Kommunikationsfunktionalitäten umfasst. Alle diese Bereiche bereits existierender Softwarelösungen werden in den folgenden Abschnitten exemplarisch behandelt.

Bei der Darstellung dieser Softwaretechnologien steht neben der Erläuterung von Begriffen zum späteren Verständnis der Softwarestrukturen auch die Verdeutlichung spezieller Softwareproblematiken im Vordergrund, die zur Auswahl verwendeter Strukturen geführt haben bzw. dazu notwendig waren.

## 2.1 Mechanismen der Systemsoftware

Mittels Systemsoftware wird Anwendungen der komfortable Zugang zu den Hardware-Ressourcen eines Rechnersystems eröffnet. Sie erfüllt damit eine wichtige Brückenfunktion zwischen Anwendungen und der Rechnerhardware und sorgt bei einem Mehrbenutzer- und Mehrprogrammbetrieb durch entsprechende Koordinierung dafür, dass sich unabhängige Anwendungsprogramme beim Zugriff auf Ressourcen des Rechnersystems nicht in die Quere kommen [4]. Die DIN 4430 beschreibt ein Betriebssystem in folgender Weise:

> „Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen“.

In der Regel bietet die Systemsoftware mit ihrem Laufzeitmodell (Abstraktion der Dienstschicht) dem Anwender und Programmierer Dienste aus folgenden *Dienstgruppen* zur Nutzung an, wobei noch weitere (interne) Dienstgruppen existieren, die die Effizienz der Systemsoftware sicherstellen:

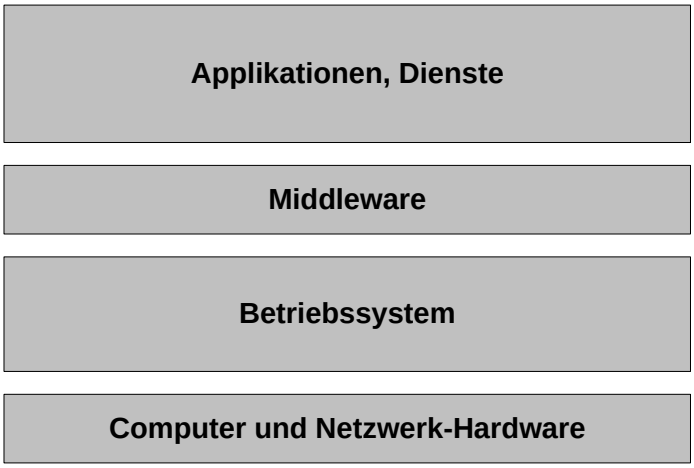



Abbildung 2.1: SW- und HW-Serviceschichten in verteilten Systemen [104]

**Systembedienung** Interaktive Führung des Benutzers, z.B. über eine Kommandoschnittstelle oder grafische Techniken zum Laden, Starten, Verwalten und Beenden von Programmen.

**Prozessmanagement** Verwaltung von Rechenaufträgen. Diese Rechenaufträge definieren Arbeitseinheiten für einen Prozessor und ermöglichen das Prozessmanagement, z.B. im Falle von Ausnahmebehandlungen oder zur Hintergrundbearbeitung nebenläufiger Prozesse. In Echtzeitsystemen sind Zeitdienste ein weiterer wichtiger Bestandteil.

**Prozessinteraktion** Unterstützung gezielter Formen des Informationsaustausches zwischen Prozessen und notwendiger Synchronisationsmechanismen (lokal oder über Kommunikationsnetze).

**Datenhaltung** Ermöglichung kurz- und langfristiger Aufbewahrung von Daten auf vorhandenen Speichermedien und des Zugriffs.

**Gerätemanagement** Komfortabler und hardwareunabhängiger Zugriff auf Geräte, z.B. über eine Ansicht in Form eines Dateisystems.

Die Dienstgruppen Prozessmanagement und Prozessinteraktion aus dem obigen Dienstkatalog stellen bei richtiger Auslegung genau die Funktionalität für die Anwendung bereit, die für den nicht-monopolisierenden, geschützten Zugriff auf die Ressourcen Prozessor und Speicher benötigt werden. Der Schwerpunkt bei der Ermittlung geeigneter Systemsoftware für Echtzeit-Anwendungen liegt also bei diesen beiden Dienstgruppen, in denen die folgenden drei Abstraktionen zu unterscheiden sind:

**Adressraum** Von der Speichertechnologie unabhängiger virtueller Speicher, der durch Speicherzellen (in der Regel als Bytes dargestellt) definiert wird. Er kann erzeugt und gelöscht werden und stellt einen Behälter für die Aufnahme von abgegrenzten Programmen mit zugehörigen Daten dar. Ein Adressraum ist immer einem

Prozess zugeordnet und in der Regel gegen den Zugriff seitens anderer Prozesse geschützt.

**Threads** Beschreiben die Aktivitäten in einem System (Aktivitätsträger). Sie können als virtuelle Prozessoren aufgefasst werden, die jeweils für die Ausführung einer speziellen Funktionalität in einem Adressraum verantwortlich sind. Sie basieren auf einem streng sequenziellen, nebenläufigen Verarbeitungsmodell und sind Prozessen zugeordnet. Gehören zwei Threads zum gleichen Prozess, so können sie auch auf die gleichen Ressourcen wie z.B. den gleichen Arbeitsraum zugreifen.

**Interaktion** Wechselwirkung zwischen zwei oder mehreren Prozessen. Dabei werden zwei Interaktionsmuster unterschieden: Erstens, Konkurrenz, bei der sich zwei oder mehrere Prozesse zeitgleich um die Nutzung derselben Ressource bewerben und somit der Zugriff für die einzelnen Prozesse zeitlich gestaffelt werden muss (Synchronisation, Scheduling); zweitens, Kooperation, bei der beteiligte Prozesse gegenseitig Informationen austauschen. Hierbei müssen sich die kooperierenden Prozesse in der Regel kennen. Sie nutzen Mechanismen aus dem Bereich der Prozesskommunikation.

Man unterscheidet bei der Interaktion von Prozessen im Zusammenhang mit Threads blockierende und nicht-blockierende Interaktionen. Blockierend bedeutet dabei, dass die Ausführung innerhalb eines Threads erst dann fortgesetzt wird, wenn ein bestimmter Systemzustand (ggf. nach einer gewissen hardwareabhängigen Wartezeit) erreicht ist. Gerade dieser Umstand ist bei Echtzeitsystemen sehr kritisch zu betrachten, da es hier darauf ankommt, definierte Maximalgrenzen für Latenzzeiten anzugeben, um zeitlichen Determinismus zu gewährleisten.

### 2.1.1 Grobarchitektur von Laufzeitsystemen

Die in den Diensten der Systemsoftware bereitgestellten Funktionalitäten stellen einen Rahmen für die Organisation von Anwendungen dar. Eine fest umrissene Funktionsmenge, die Anwendungsprogrammierern und Benutzern während der Laufzeit zur Verfügung steht, wird als Laufzeitsystem bezeichnet [4]. Man unterscheidet verschiedene Softwarearchitekturen für Laufzeitsysteme, die sich nach und nach den programmiertechnischen Anforderungen der Soft- und Hardwaretechnik angeglichen haben. Nachfolgend sind die wichtigsten Meilensteine in der Entwicklung von Softwarearchitekturen für Laufzeitsysteme skizziert [5] [9]:

**Monolithisch** Das Laufzeitsystem ist gemeinsam mit den Anwendungsprogrammen eine Sammlung von Prozeduren und Funktionen ohne prinzipielle Struktur, bei der jede Prozedur eine beliebige andere Prozedur aufrufen und beliebige Datenstrukturen modifizieren kann. Funktionen zur Realisierung gemeinsamer Teilfunktionen werden oftmals gruppiert, sind aber nicht streng gekapselt.

**Geschichtet** Das Laufzeitsystem und die Anwendungsprogramme sind in mehreren Schichten organisiert, wobei die Schicht n+1 die Funktionen der untergeordneten Schichten 1,...,n nutzt, um ihre eigenen Funktionen zu realisieren. Die

verwendete modulare Struktur erleichtert zwar schon die Fehlerlokalisierung, allerdings ist eine Festlegung der Funktionalitäten für die einzelnen Schichten schwierig. Auch erfordert die Sicherstellung der strikten Einhaltung dieser Schichtung und die strenge Parameterprüfung bei Funktionseintritt einen erheblichen Verwaltungs-Overhead.

**Mikrokernel** Das Laufzeitsystem ist so strukturiert, dass alle nicht-essenziellen Komponenten aus dem Systemkern herausgenommen und in Form von System- oder Anwenderprogrammen implementiert sind. Der Kern wird dadurch sehr klein und beinhaltet selbst nur noch die nötigsten Funktionalitäten des Prozess- und Speichermanagements sowie der Kommunikation.

**Modul** Das Laufzeitsystem beinhaltet objektorientierte Programmiermethoden, um einen modularen Systemkern zu erzeugen. Der Systemkern besitzt einen Satz von essentiellen Komponenten und bindet beim Hochfahren oder während der Laufzeit des Systems zusätzliche Dienstmodule dynamisch ein. Jede Zugriffssektion hat dabei eine fest definierte Schnittstelle. Der Zugriff ist damit ähnlich der der geschichteten Architektur, mit dem Unterschied größerer Flexibilität, da jedes Modul auf ein beliebiges anderes Modul zugreifen kann.

Die Dienste des Laufzeitsystems können grundsätzlich auf drei verschiedene Weisen bereitgestellt werden. Abbildung 2.2 zeigt die Grobarchitektur eines solchen Laufzeitsystems bei einer Bereitstellung der Dienste im Adressraum der Anwendung (1), in einem abgeschlossenen Kern (2) und in Servern, die in separaten Adressräumen liegen (3). Der Zugriff auf solch einen Dienst ergibt sich immer durch Aufruf einer Funktion im Adressraum der Anwendung. Die Funktion kann den Dienst dabei also entweder selbständig erbringen oder dazu die Unterstützung des Kerns bzw. anderer Server in Anspruch nehmen. Je nachdem welche zeitlichen oder funktionalen Anforderungen für die spezifische Funktion gelten, werden sie in einem der drei Bereiche implementiert.

Um den Umfang von Laufzeitsystemen möglichst zu minimieren und gleichzeitig auf Funktionalitäten über ein Netzwerk zugreifen zu können, werden Teilfunktionen des Laufzeitsystems in spezielle Server ausgelagert. Das Laufzeitsystem selbst übernimmt damit vor allem die Vermittlung der Systemaufrufe des Anwenders an den entsprechenden Server (Kommunikationsprimitive) und das elementare Multi-Programming. Dadurch, dass die Server im Benutzermodus ablaufen können, erhält das Laufzeitsystem selbst eine gesteigerte Laufzeitstabilität. Diese **Client/Server**-Zugriffsorganisation wird oft sowohl für die Systemsoftware als auch für die Anwendungsprogramme verwendet.

### 2.1.2 Prozess-Organisation

Bei Verwendung einer nebenläufigen Softwarearchitektur mit Threads ist es notwendig, eindeutig zu bestimmen, zu welchem Zeitpunkt welcher der im System vorhandenen Threads den physischen Prozessor gerade besitzt und festzulegen, aufgrund welcher Bedingungen ein Wechsel (Kontextwechsel) zu einem anderen Thread erfolgt. Zu diesem Zweck besitzt jeder Thread eine eindeutige Prozessidentifikationsnummer

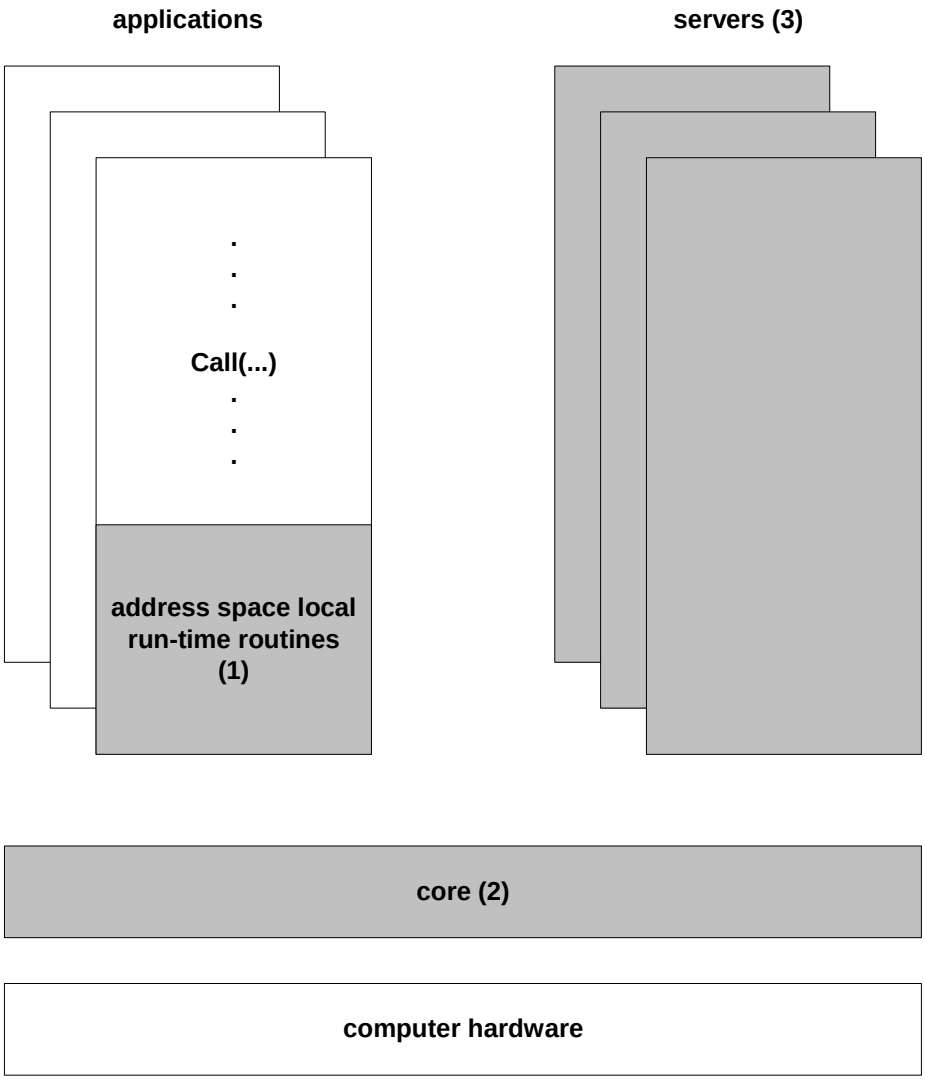



Abbildung 2.2: Grobarchitektur eines Laufzeitsystems

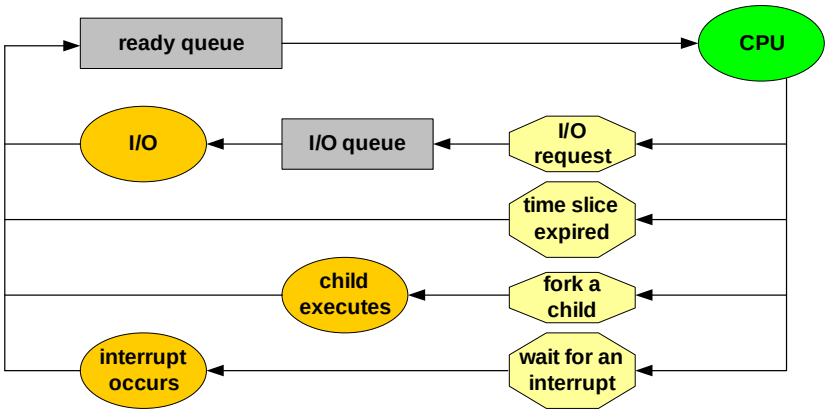



Abbildung 2.3: Organisation des Schedulings über Warteschlangen

und Zustandsinformation, auf deren Basis eine Neustrukturierung innerhalb des Softwaresystems erfolgen kann.

Die Durchführung der Kontextwechsel (Wechsel von einem aktiven Thread zum nächsten) erfolgt über einen Dispatcher und einen Scheduler. Der Dispatcher sorgt für einen effektiven Wechsel des Kontextes zweier sich ablösender Threads. Dabei übernimmt er die Speicherung des Prozessorstatus, Sicherung von Ressourcen und die Aktivierung eines neuen Prozessorstatus. Der Scheduler legt die Auswahl eines neuen Threads aus der wartenden Threadmenge gemäß gewählter Strategie (FCFS[1] , RR[2] , SJF[3] etc.) fest [5]. Welche der präemptiven oder nicht-präemptiven Schedulingstrategien für den einzelnen Thread ausgewählt wird bzw. in welcher Kombination sie gewählt werden, hängt von den Möglichkeiten der Systemsoftware und den gewünschten Leistungseigenschaften ab. Beim Scheduling wird über die gewählten Schedulingstrategien versucht, die Handhabung der Prozesse und Threads in der Anwendung so zu gestalten, dass folgende Zielkonflikte für den Benutzer oder das System möglichst optimal gelöst werden (dabei erhebt die Liste keinen Anspruch auf Vollständigkeit) [6]:

- Auslastung der CPU
- Anzahl der Threads pro Zeiteinheit (Durchsatz)
- Ausgeglichenheit bei der Vergabe der CPU an laufende Threads
- Zeitspanne zwischen Start und Ende eines Threads
- Wartezeit bis zur Ausführung eines Threads
- Reaktionszeit des Systems nach einem externen Ereignis

[1] FCFS = First-Come-First-Serve - Sequentielle Abarbeitung der Threads gemäß Bereitschaft
[2] RR = Round-Robin - Threads erhalten einen definierten Zeitschlitz zur Prozessornutzung
[3] SJF = Shortest-Job-First - Kurze Threads erhalten Vorrang bei der Prozessorzuteilung

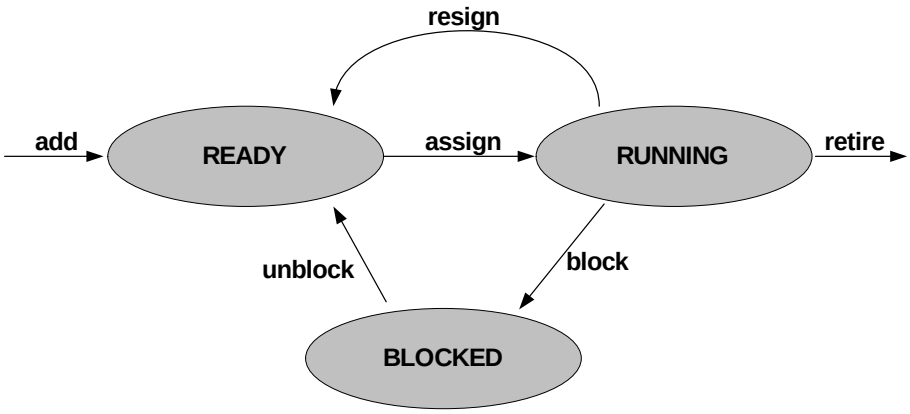



Abbildung 2.4: Grundlegendes Thread-Zustandsmodell

Abbildung 2.3 zeigt den prinzipiellen Aufbau eines CPU-Schedulers, der für das kurzfristige Verteilen der Prozessorressourcen auf die im System vorhandenen Threads verantwortlich ist. Die grauen Kästchen stellen Listenstrukturen dar, in denen die jeweiligen Threadinformationen gemäß ihrer aktuellen Priorität gespeichert sind. Wird nun ein Thread aus der Ready-Queue vom Scheduler aktiviert, so wird dieser von der CPU so lange ausgeführt, bis er ein Ein- oder Ausgabeereignis der Hardware anfragt, seine Zeitscheibe abgelaufen ist, einen anderen Thread erzeugt oder auf das Eintreffen eines Interrupt-Ereignisses wartet (gelbe Vielecke). Wenn das entsprechende in den orangen Ovalen beschriebene Ereignis eingetroffen ist, gelangt der Thread wieder in die Ready-Queue und wird dann gemäß seiner Priorität und aktiven Schedulingstrategie zu gegebenem Zeitpunkt wieder aktiviert.

### Zustandsmodell

Als Grundlage zur Verteilung der Prozessorleistung auf die verschiedenen Threads innerhalb eines Softwaresystems dient ein Thread-Zustandsmodell (vereinfacht in Abbildung 2.4 dargestellt). Jeder aktive Thread im System befindet sich zu jedem Zeitpunkt in einem der dargestellten möglichen Zustände. Ein Wechsel in einen anderen Zustand geschieht jeweils nach dem Hinzufügen und Beenden eines Threads, beim Eintreffen einer externen Unterbrechungs-Anforderung (Interrupt) oder beim Aufruf einer Funktion aus der Systemsoftware-Bibliothek. Es werden dabei grundsätzlich diese drei zur Laufzeit wesentlichen Zustände unterschieden:

**RUNNING** Threads in diesem Zustand sind im Besitz des physischen Prozessors. Bei Ein-Prozessor-Systemen trifft das nur für genau einen Thread für pro Zeitpunkt zu. In diesen Zustand gelangt ein Thread nur nach erfolgreichem Neu-Scheduling aus dem READY-Zustand.

**BLOCKED** Blockierte Threads warten auf das Eintreten einer spezifischen Systembedingung (externe Ereignisse, Timer, Synchronisation oder Variablen). Bedingt durch das Scheduling innerhalb der Systemsoftware, das Eintreffen externer Unterbrechungs-Anforderungen oder den Aufruf einer Systemfunktion kann aus

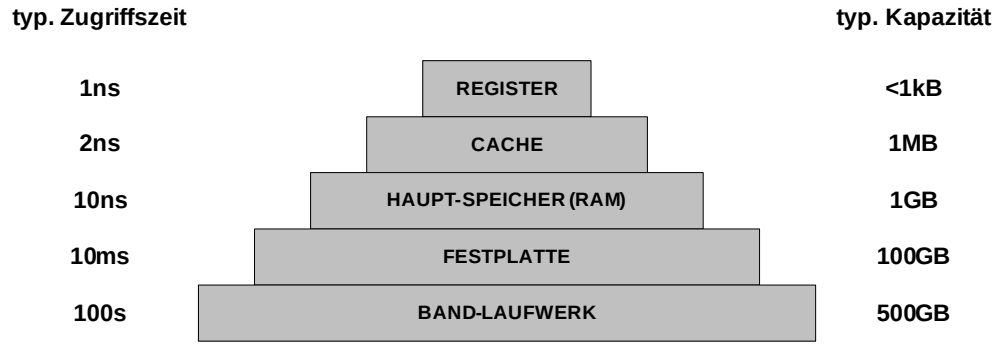



Abbildung 2.5: Typische Speicherkonfigurationen

dem RUNNING-Zustand in diesen Zustand gelangt werden. Es können sich zu jeder Zeit beliebig viele Threads in diesem Zustand befinden.

**READY** Threads in diesem Zustand sind prinzipiell ausführbar, aber momentan nicht im Besitz des physischen Prozessors. Sie können aus dem BLOCKED-Zustand durch Eintreffen der erwarteten Bedingung in diesen Zustand gelangen oder aus dem RUNNING-Zustand durch Verdrängung während des Schedulings. Bei einem neuen Scheduling wird aus allen Threads im READY-Zustand höchstpriore Thread zur Zuordnung des physischen Prozessors ausgewählt. Im READY-Zustand können sich zu jeder Zeit beliebig viele Threads befinden.

Um den Ablauf von Threads in einem System steuern zu können, besitzt jeder Thread detaillierte Informationen im zugehörigen „process control block“ (PCB) innerhalb der Systemsoftware, in dem neben dem Thread-Zustand auch Prozessor-Register-Inhalte, Prioritäten und Daten zur Verwaltung von Hauptspeicher und Betriebsmitteln gespeichert sind.

### Speicherverwaltung

In einem Computersystem werden unterschiedliche Speicherarten verwendet, um darin die Arbeitsprozesse selbst und die für die Prozessbearbeitung notwendigen Daten abzulegen (Abbildung 2.5). Dabei ist der Speicher hierarchisch organisiert und weist für den jeweiligen Anwendungsbereich passende Eigenschaften auf. Die Verwendung von (schnellen) Registern für die Berechnung von Prozessdaten ist für den Anwender nur bedingt wählbar, da die Nutzung typischerweise dem Prozessor selbst vorbehalten ist. Die Kosten und der Platzbedarf steigen hierbei mit wachsender Datengröße ungünstig an; ein direktes Ansprechen auf Hochsprachenebene ist umständlich. Der Cache-Speicher ist zwar noch relativ schnell, birgt allerdings einige Synchronisierungstücken hinsichtlich der gemeinsamen Nutzung externer Speichermedien in sich. Der große Hauptspeicher ist für die meisten Anwendungen eine passende Alternative zu den schnelleren Speicherarten. In ihm werden alle Anwenderprogramme und Daten zur Laufzeit der Prozesse aufbewahrt. Langsamere Speichermedien wie Festplatte und Bandlaufwerk werden zur Speicherung und Archivierung von Programmen, im Allgemeinen aber nicht bei der Ausführung zeitkritischer Anwendungen genutzt.

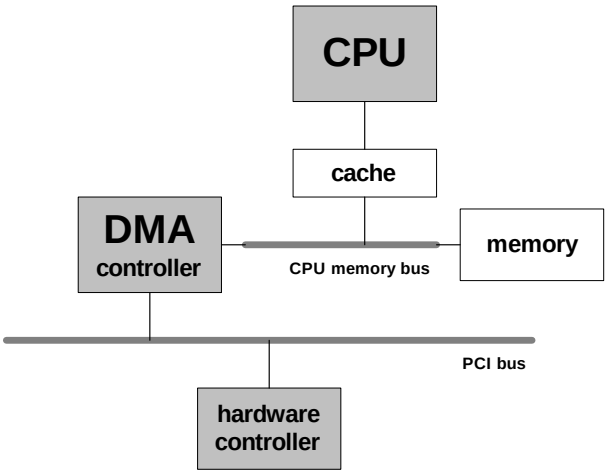



Abbildung 2.6: Schema für Direct-Memory-Access

Ein Teil der Systemsoftware, die Memory-Management-Unit (MMU), organisiert Vergabe, Belegung, Zugriff und Freigabe von Speicher für alle Anwender innerhalb eines Computersystems. Beispielsweise fallen bei der Verwendung des Cache-Speichers im Zusammenhang mit dem Verschieben von Daten zwischen zwei Bereichen im Hauptspeicher Aufgaben an, die im günstigen Fall eine Beschleunigung weiterer Speicherzugriffe ermöglichen können. Eine weitere Aufgabe des Memory-Managements ist das Auslagern („Swapping“) von Daten und Prozessen aus dem Hauptspeicher auf ein langsameres Speichermedium, wenn der Hauptspeicher ausgelastet ist. Auch die Organisation des Virtuellen-Speicher-Mechanismus, der mittels „Paging“ physikalische Speicher-Raum-Beschränkungen umgehen kann, gehört zu den Aufgaben der MMU. Diese Mechanismen sind im Allgemeinen mit großen Verzögerungszeiten bei der Prozessausführung verbunden, so dass sorgfältige Überlegungen zu ihrer Nutzung im Zielsystem angestellt werden müssen.

### Speicherzugriff über DMA

In vielen Prozessorsystemen besteht die Möglichkeit, externer Hardware einen direkten Zugriff auf den Arbeitsspeicher zu geben, um große Speicherblöcke effektiv und selbständig zu übertragen, ohne den Prozessor dabei aktiv zu belasten. Dieses Verfahren wird als Direct Memory Access (DMA) bezeichnet und steht im Gegensatz zu dem programmierten Ein-Ausgabe-Verfahren (PIO), bei dem der allgemeine Prozessor zur Übertragung eines Datenblocks von der bzw. an die Hardware jedes einzelne Datenwort aktiv überträgt. Zur Realisierung von DMA existiert eine zusätzliche Kontrollhardware, die Arbeitsaufträge zum Datentransfer von der CPU erhält und Daten beim Eintreffen direkt an die gewünschte Adresse im Arbeitsspeicher ablegt bzw. sie dort abholt und an die entsprechende Hardware überträgt. Beim Datentransfer wird die CPU nicht beansprucht; sie erhält lediglich einen automatischen oder erfragten Statusbericht über den Fortschritt des Transfers.

Abbildung 2.6 zeigt die am DMA beteiligten Komponenten des Prozessorsystems. Da der Arbeitsspeicher (Memory) in der Regel nicht als Dual-Port-RAM ausgeführt ist,

kann darauf nur immer eine angeschlossene Komponente pro Zeiteinheit zugreifen. Dies bedeutet wiederum, dass die CPU (bei entsprechend gewählter Priorisierung) für die Zeit des aktuellen DMA-Speicherzugriffs selbst nicht auf den Speicher zugreifen kann. Solange von dort aber keine Zugriffe auf den Arbeitsspeicher notwendig sind, sondern zur weiteren Programmabarbeitung auf den Cache-Speicher zugegriffen werden kann, kommt es dabei zu keiner merklichen Verzögerung im Programmablauf. Zur Sicherstellung kürzester Prozessor-Reaktionszeiten trotz der Verwendung externer Hardwarekomponenten ist es daher angebracht, die Nutzung des Cache-Speichers applikationsspezifisch zu optimieren.

### Pufferung als Design-Eigenschaft für geschichtete Software

Ein Vorteil geschichtet aufgebauter Softwaresysteme ist ihre klare Struktur, die einen sicheren und einheitlichen Zugriff auf schichtinterne Daten und Methoden ermöglicht. Ein weiterer Vorteil ist die Möglichkeit, einzelne Schichten zu ersetzen oder zu modifizieren, ohne dabei den Zugriff darauf (API[4]) zu verändern. Bei Anwendungen, die auf großen Datendurchsatz und dabei möglichst kleine Reaktionszeiten angewiesen sind, führt eine konsequent realisierte Schichtung in der Regel zu Schwierigkeiten. Besonders im Bereich der Robotersteuerungen benötigen die oberen Softwareschichten (Roboterprogrammierung und Steuerung) Sensordaten, die in der untersten Softwareschicht (Kommunikation) bereitgestellt werden. Die Schichtung bedingt eine Pufferung jeglicher Daten, die zwischen den einzelnen Schichten kommuniziert werden sollen. Je mehr Schichten ein Softwaresystem zu seiner Realisierung verwendet, umso öfter werden die Daten, die durch die Schichten kommuniziert werden müssen, intern kopiert, bis sie in der Zielschicht ankommen. Diese Schichtung muss daher bei der Realisierung reaktionsschneller Anwendungen an geeigneter Stelle auf Kosten der klaren Struktur aufgebrochen werden, um einen direkten Datenzugriff (beispielsweise über Zeiger) ermöglichen zu können.

### Interprozesskommunikation

Prozesse, die mehr oder weniger gleichzeitig in einem Rechnersystem ablaufen, sind entweder unabhängig voneinander oder kooperativ. Im kooperativen Fall tauschen Prozesse Daten miteinander aus, um gemeinsam bei der Erfüllung einer Aufgabe beteiligt zu sein. Dabei existieren in Betriebssystemen zwei grundsätzlich verschiedene Mechanismen: Shared-Memory und Message-Passing.

Mit Hilfe von **Shared-Memory** können statische Produzenten/Konsumenten-Verhältnisse zwischen Prozessen aufgebaut werden, die einen schnellen Datenaustausch ermöglichen. Abbildung 2.7 zeigt die grundsätzliche Struktur der Shared-Memory-Nutzung. Zwei Prozesse, die jeweils über einen eigenen Arbeitsraum (hellgraue Ellipsen) und ein oder mehrere Threads (Rechtecke) verfügen, vereinbaren einen dritten Arbeitsraum, der sich aus der Überschneidung der einzelnen Arbeitsräume definiert. Auf diesen Shared-Memory-Bereich können nun die Threads beider Prozesse gleich-

[4] API = Application Programmer Interface

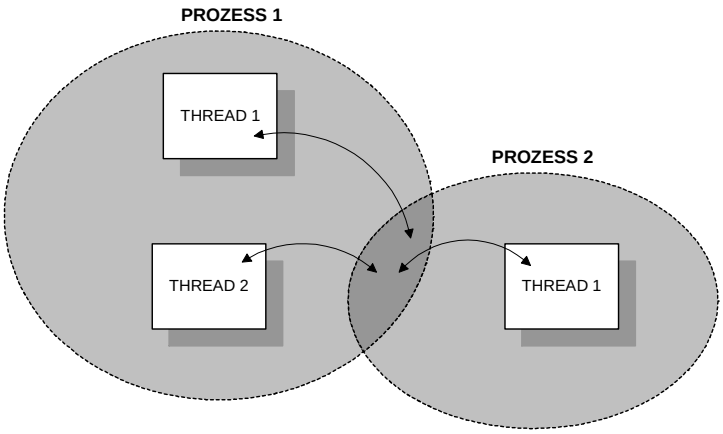



Abbildung 2.7: Schema der Shared-Memory-Nutzung

berechtigt und wahlfrei zugreifen. Die Sicherstellung der Datenkonsistenz wird dabei aber nicht vom Betriebssystem übernommen. So kann es vorkommen, dass ein Thread des ersten Prozesses zu einem Zeitpunkt unterbrochen wird, zu dem das Schreiben der Daten in den Shared-Memory noch nicht vollständig abgeschlossen ist. Kommt nun aber der Thread aus dem zweiten Prozess zur Ausführung und greift lesend auf dieselben Daten zu, so führt dies sehr wahrscheinlich zu Fehlinterpretationen der gelesenen Daten. Abhilfe hierbei schafft die Verwendung von Synchronisationsmechanismen, die vom verwendeten Betriebssystem angeboten werden müssen und weiter unten erläutert sind.

Während das Betriebssystem die Shared-Memory-Bereiche nur verwaltet und der Anwenderprozess für die konsistente Nutzung verantwortlich ist, wird der zweite Kommunikationsmechanismus, **Message-Passing**, direkt und komplett über das Betriebssystem realisiert. Prozesse können so mit anderen Prozessen kommunizieren und gleichzeitig ihre Aktivitäten (innerhalb der Threads) miteinander synchronisieren, ohne dabei den gleichen Adressraum zu verwenden. Dadurch ist das Message-Passing auch für Netzwerke mehrerer Rechner interessant. Abbildung 2.8 zeigt hierzu ein prinzipielles Funktionsschema. Das Message-Passing stellt zu seiner Realisierung grundsätzlich zwei Operationen (Funktionen) zur Verfügung: *send* und *receive*. Im Allgemeinen muss dabei vor der Ausführung dieser Operationen bereits eine Kenntnis der Identitäten der Kommunikationspartner vorliegen. Folgende Arten der Referenzierung werden unterschieden [9]:

**Direkte Kommunikation** Jeder Kommunikationspartner referenziert seinen Bezugsprozess über den jeweiligen Prozessnamen bzw. die zugehörige Identifikationsnummer (Prozess-ID) innerhalb des Funktionsarguments. Dabei wird bei der Verbindung zwischen genau zwei Kommunikationspartnern von *symmetrischer* Adressierung gesprochen. Eine Variation dieser Kommunikationsart ist die Verwendung einer variablen Sender-ID in der Empfangsfunktion, die beim Empfang auf die tatsächliche Sender-ID gesetzt wird. Auf diese Weise kann der Empfänger auf mehrere Sender-Prozesse reagieren (*asymmetrische* Adressierung).

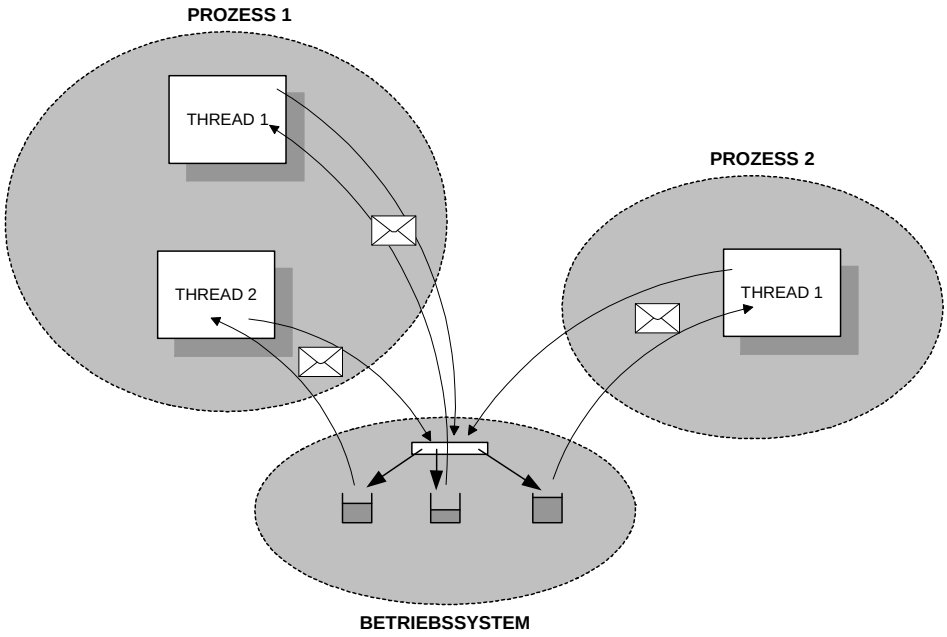



Abbildung 2.8: Schema der Message-Passing-Nutzung

**Indirekte Kommunikation** Das Senden sowie das Empfangen von Nachrichten geschieht über Postkästen (Mailbox), die entweder dem Empfängerprozess zugeordnet werden oder aber dem Betriebssystem selbst. Kommunizierende Prozesse vereinbaren den Zugriff auf solch einen Postkasten. Dieser kann dann Nachrichten von verschiedenen Prozessen enthalten, die alle von einem Prozess ausgewertet werden.

Beim Message-Passing versendet oder empfängt ein Prozess Nachrichten, indem die Nutzdaten über das Betriebssystem von einem Prozess-Adressbereich in den anderen kopiert werden. Dabei kann dieser Kopiervorgang für die beteiligten Prozesse entweder *blockierend* (synchron) oder *nicht-blockierend* (asynchron) sein:

**Blockierendes Senden** Der sendende Prozess ist mindestens so lange blockiert, bis die Nachricht beim Empfänger ankommt. In manchen Spezialisierungen oder Implementierungen sogar so lange, bis eine erwartete Antwort wieder beim Sender der Nachricht ankommt.

**Nicht-blockierendes Senden** Der sendende Prozess setzt die Nachricht einfach ab und arbeitet weiter. Die Nachricht wird, sofern der Empfänger sie nicht sofort empfangen kann, in einem Postkasten aufgehoben, bis der Empfänger ein *receive* ausführt.

**Blockierendes Empfangen** Der Empfänger blockiert so lange, bis eine Nachricht für ihn eintrifft. Falls der Postkasten noch keine entsprechende Nachricht enthält, wartet der Empfänger, bis der Sender die Nachricht abgesetzt hat und führt dann seine Bearbeitung fort.

**Nicht-blockierendes Empfangen** Der Empfänger prüft, ob eine Nachricht für ihn vorliegt. Ist das der Fall, wird die Ausführung unter Berücksichtigung der Nachricht fortgeführt; sonst wird angezeigt, dass keine Nachricht vorlag und der Prozess läuft weiter.

Über die Funktionsparameter sind je nach beabsichtigtem Anwendungsfall unterschiedliche Kombinationen der gewünschten Sende- und Empfangssynchronisation wählbar. Allerdings treten bei Message-Passing unter Verwendung blockierender Sende- und Empfangstransfers, anders als bei der Nutzung von Shared-Memory, keine Datenkonsistenz-Schwierigkeiten mehr auf.

### Prozess-Synchronisation

Eine Systemsoftware muss dem Anwender über eine geeignete Programmierschnittstelle Mechanismen zur Verfügung stellen, die es den Prozessen und Threads ermöglichen, zur Laufzeit Vorrangbeziehungen untereinander festzulegen und eine gegenseitige Synchronisation durchzuführen. Die elementaren betriebssystemgestützten Mechanismen speicherbasierter Prozessinteraktion sind hier aufgeführt:

**Mutex** Mutex steht für Mutual Exclusion und bezeichnet einen Mechanismus, der es Prozessen und Threads ermöglicht, den Zugriff auf eine gemeinsam genutzte Ressource (Speicher, Variablen, Geräte etc.) wechselseitig auszuschließen. Dabei ist ein Mutex als einen Schlüssel vorstellbar, den nur genau ein Benutzer pro Zeiteinheit besitzen darf und nach der Benutzung der Ressource wieder abgeben muss.

**Semaphore** Semaphoren ermöglichen es, eine bestimmte Anzahl von Prozessen oder Threads für die Nutzung einer Ressource zuzulassen. Sie stellen damit in gewisser Weise eine Erweiterung des Mutex-Mechanismus dar, der die gleichzeitige Vergabe von mehreren Schlüsseln ermöglicht.

**Condition Variable** Eine Condition Variable ist eine von einem Anwender definierbare und veränderbare Bedingungsvariable. Wenn die gewünschte Bedingung noch nicht eingetreten ist, wartet der darauf synchronisierte Thread (blockiert). Jede Änderung der Bedingungsvariablen kann den Thread aufwecken, so dass dieser die Bedingung erneut abprüft. Auf diese Weise können einfach synchronisierte sequentielle oder parallele Folgebearbeitungen z.B. eines Producer-/Consumersystems realisiert werden.

**Monitore** Ein Softwarekonstrukt, das die Mechanismen Mutex und Condition Variable nutzt, um eine für den Benutzer einfach handhabbare Form von bedingungsabhängigem Schutz „kritischer Bereiche“ zu realisieren. Der Monitor beugt Konfliktsituationen vor.

Die praktische Anwendung der hier aufgezählten elementaren Mechanismen wird bei der Beschreibung der Realisierung der in dieser Arbeit entstandenen Software detailliert beschrieben.

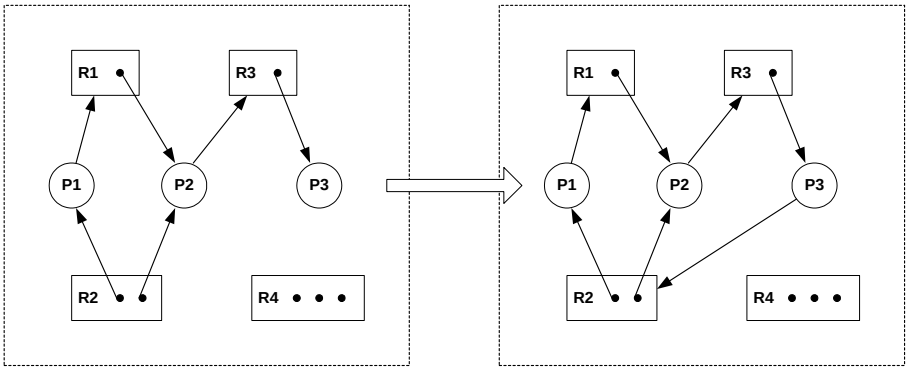



Abbildung 2.9: Resource-Allocation-Graphen

### Synchronisationsfehler

In interagierenden Prozesssystemen können Synchronisationsfehler in Programmabläufen Ablauf-Inkonsistenzen oder Verklemmungen hervorrufen. Bei einer Ablauf-Inkonsistenz handelt es sich um das zeitliche Nicht-Abgestimmt-Sein von Dateninteraktionen zwischen Threads. Diese zeigen sich in u.U. sehr schwer reproduzierbaren Daten-Inkonsistenzen. Eine Verklemmung ist ein Zustand, in dem die beteiligten Threads wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Threads dieser Gruppe selbst hergestellt werden können. Als Folge dieser wechselseitigen Abhängigkeit bleiben die beteiligten Threads permanent blockiert. Verklemmungen sind als einzige Form zeitabhängiger Fehler beschreibbar und formalisierbar. Mittels eines einfachen Graphenmodells nach Holt [8] lassen sich für interagierende Prozesssysteme präzise definierte Prozesszustände beschreiben, auf die später Erkennungs- oder Vermeidungsalgorithmen angewandt werden können. Allerdings sind diese Algorithmen bei größerer Anzahl beteiligter Threads wegen der kombinatorischen Anzahl an Systemzuständen nicht mehr beherrschbar. Eine weitere Möglichkeit, systematische Synchronisationsfehler zu detektieren ist es, die Zuteilung von Betriebsmitteln zu untersuchen. Dazu hilft z.B. der Resource-Allocation-Graph [9], bei der die drei Mengen Prozesse (p), Betriebsmittel (r) und Anforderungs-/Benutzer-Relation (Pfeile) unterschieden werden.

Abbildung 2.9 zeigt zwei solcher Resource-Allocation-Graphen, die einen Übergang von einem verklemmungsfreien zu einem verklemmten Systemzustand darstellen. Notwendige, aber nicht hinreichende Bedingung für eine Verklemmung ist das Vorhandensein von gerichteten Relationszyklen, in diesem Fall beispielsweise:

p1 -> r1 -> p2 -> r3 -> p3 -> r2 -> p1 und p2 -> r3 -> p3 -> r2 -> p2 .

Erkennungs- und Vermeidungsalgorithmen nutzen die zirkulare Wartebedingung als allgemeines Kriterium und als Ausgangspunkt für weitere Untersuchungen zur Ermittlung sowie Vermeidung von Verklemmungen (z.B. [10] und [11]). Allerdings sind diese Algorithmen in der Praxis nicht einsetzbar, da sie für allgemeine Systeme mit u.U. vielen Threads und Zuständen zu rechenintensiv sind. Aktuelle Systemsoftware bietet allerdings programmiertechnische Hilfsmittel (Semaphoren, Mutexe, etc.) an, um diese Überwachungs- und Vermeidungsproblematik auf Benutzerebene zu lösen.

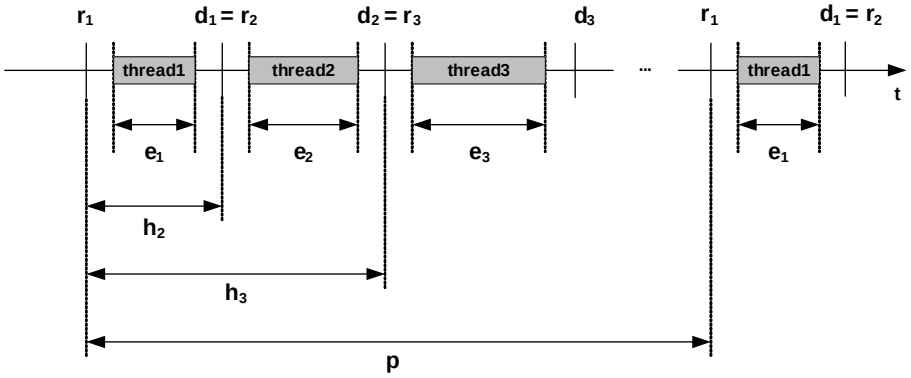



Abbildung 2.10: Zeit-Definitionen für Echtzeit

**Synchronisation ohne Systemsoftware**

Bei den in dieser Arbeit entstandenen Softwareimplementierungen aus dem Kapitel 4 wird für die Realisierung auf der DSP-Plattform kein eigenes Betriebssystem verwendet, da die Zusammenhänge aufgrund der funktionalen Beschränkungen in ausreichendem Maße überschaubar sind. Es wurde bei der Implementierung der Hardwaretreiber der untersten Softwareebene besonders darauf geachtet, dass die Schnittstelle zu den übergeordneten Softwareschichten im Vergleich zu der Implementierung unter QNX identisch ist; also der Benutzer weder vom zeitlichen Verhalten noch vom syntaktischen Zugriff her Unterschiede feststellt. Die in diesem Kapitel gemachten Aussagen zur Echtzeitfähigkeit sind besonders berücksichtigt worden, da auf Grund des fehlenden Betriebssystems die Mechanismen zur Sicherstellung von Echtzeitfähigkeit „von Hand“ implementiert werden mussten.

### 2.1.3 Echtzeitfähigkeit von Anwendungen

Robotersteuerungen und auch sicherheitsrelevante Systeme innerhalb von Fahrzeugen sind dadurch gekennzeichnet, dass ihre korrekte Funktion nicht nur vom korrekten Ergebnis logischer Berechnungen abhängt, sondern auch von den zeitlichen Bezügen, in denen diese Ergebnisse produziert werden. Eine wesentliche Eigenschaft ist hierbei also, dass Zeitvorgaben (z.B. Reaktionszeiten von Prozessen oder Threads) unbedingt eingehalten werden müssen. Diesem Schedulingziel werden alle anderen Aspekte untergeordnet. Die Zeitvorgaben sind dabei von der Anwendung definiert. Sie sind unabhängig davon, ob die Anwendung mit Zykluszeiten im Stunden-, Sekunden- oder Nanosekundenbereich arbeitet. Man unterscheidet zwischen „harter“ oder „strikter“ Echtzeitfähigkeit, bei der eine Verletzung der definierten Zeitvorgaben nicht tolerierbar ist [9] und „weicher“ oder „schwacher“ Echtzeitfähigkeit, bei der sich eine Verletzung der Zeitbedingungen maximal in störenden, nicht aber in systemkritischen Effekten äußert. Abbildung 2.10 zeigt eine allgemeine Form eines Zeitdiagramms für Echtzeit-Threads. Jeder Thread ist dabei durch drei Kenngrößen charakterisiert:

- den frühestmöglichen Startzeitpunkt ($r_x$ - READY TIME),

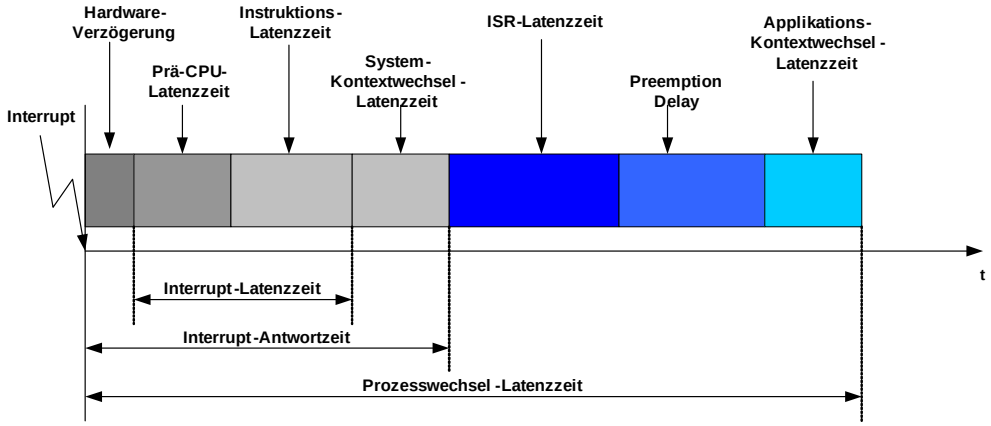



Abbildung 2.11: Latenzzeiten

- die zur Verfügung stehende zeitliche Frist ($d_x$ - DEADLINE) und
- die maximale Ausführungszeit ($e_x$ - EXECUTION TIME).

Vor dem jeweiligen frühestmöglichen Startzeitpunkt (Bereitzeit) darf der betreffende Thread nicht gestartet werden, da z.B. notwendige Informationen zur Bearbeitung noch nicht vorliegen. Die Frist beschreibt die einzuhaltende Zeitvorgabe. Die maximale Ausführungszeit ist die Zeit, die im längstmöglichen Fall benötigt werden darf, z.B. wenn die aufwändigsten Teile des implementierten Codes zur Ausführung kommen (worst execution time). Die Bestimmung dieser Zeit ist in vielen Fällen sehr aufwändig. Häufig werden Echtzeitthreads periodisch (in festen zeitlichen Abständen) ausgeführt. Kenngrößen sind hierbei die Periode ($p$), die der Kehrwert der Zyklusfrequenz ist, und die Phase ($h_x$), die den Versatz des Ausführungsbeginns (Bereitzeit) relativ zum Anfang der Periode angibt.

### Latenzzeiten

Alle hier erwähnten zeitlichen Kenngrößen müssen unter Berücksichtigung der Zeiten, die während der Laufzeit zur Durchführung der Kontextwechsel und anderer Systemfunktionen anfallen, festgelegt werden. Diese Zeiten sind für jede Systemsoftware unterschiedlich und dazu noch stark abhängig vom verwendeten Prozessortyp, seiner Verarbeitungsgeschwindigkeit, der Rechnerarchitektur und der momentanen Systemauslastung durch nebenläufige Applikationen. Eine genaue rechnerische Bestimmung dieser Zeiten ist daher unmöglich, allerdings können experimentell Schätzwerte ermittelt werden.

Abbildung 2.11 verdeutlicht die allgemeine Bestimmung von Latenzzeiten. Ein Interruptvorgang wird z.B. über eine Hardware oder den Systemtimer angestoßen. Nach einer gewissen Hardwareverzögerung (hervorgerufen durch Gatterlaufzeiten) gelangt das Interruptsignal an den Interruptcontroller, der anhand der für diesen Interrupt festgelegten Priorität ermittelt, ob eine sofortige Weiterleitung an die CPU erfolgen soll oder nicht. Die (allgemein unbestimmte) Zeit, bis zu der der eintreffende

Interrupt als der höchstpriore erkannt ist, heißt Prä-CPU-Zeit. Ist der Interrupt zur CPU durchgereicht, muss zunächst der aktuelle Befehlszyklus beendet werden. Die dafür benötigte Zeit heißt Instruktions-Latenzzeit und dauert bei CISC[5]-Systemen länger als bei RISC[6]-Systemen. Nach dem Ende des Befehlszyklus erfolgt während der System-Kontextwechsel-Latenzzeit eine Kontrollübergabe an die Interrupt-Service-Routine (ISR). Hierbei werden alle zum Rücksprung nötigen Parameter und Register auf dem Stapelspeicher abgelegt. Die Abarbeitung der ISR erfolgt innerhalb der ISR-Latenzzeit, die dann die Applikationsroutine - z.B. über ein Signal - benachrichtigt, aber erst später startet. Nach Beendigung der ISR kann das System nun gemäß des gewählten Schedulingalgorithmus einen Kontextwechsel zum vorher unterbrochenen Prozess oder zu einem priorisierten Prozess (z.B. zur neuen Applikationsroutine) durchführen. Die Verzögerung zur Ermittlung des höchstprioren Prozesses (im Scheduler) wird Preemption-Delay genannt. Hiernach erfolgt innerhalb der Applikations-Kontextwechsel-Latenzzeit die Kontrollübergabe an die Applikationsroutine, die mit der ursprünglichen Interruptquelle verknüpft war. In der Abbildung finden sich auch die allgemeinen Definitionen von Interrupt-Latenzzeit, Interrupt-Antwortzeit und Prozesswechsel-Latenzzeit im Zusammenhang mit den oben beschriebenen Zeiten.

### Systemmodelle

Prinzipiell werden beim Echtzeitscheduling zwei Systemmodelle zur Ausführung von Echtzeitanwendungen unterschieden. Bei der zeitgesteuerten Ausführung basieren die Scheduling-Entscheidungen auf dem Voranschreiten der Zeit. Dabei wird, ausgehend von Timer-Interrupts (Ticks), zu jedem vorher festgelegten Zeitpunkt eine Scheduling-Entscheidung getroffen und der entsprechende Thread gestartet (und ggf. wieder beendet). Die ereignisgesteuerte Ausführung dagegen leitet die Scheduling-Entscheidung aus möglicherweise komplex zusammenhängenden internen und externen Ereignissen ab. In komplexen, dynamischen Echtzeitanwendungen kann das Systemmodell auch auf einer Kombination beider hier erwähnten Grundmodelle basieren.

Anwendungen mit harten Echtzeitanforderungen werden in der Regel vor der Programmausführung modelliert und simuliert, um sicher zu gehen, dass die Zeitvorgaben garantiert eingehalten werden. Die Schedulingstrategie und die Ausführungszeitpunkte werden offline festgelegt und entweder beim Kompilieren des Programms fest in den Code übernommen oder zur Laufzeit in Form einer Tabelle an den Scheduler übergeben, der dann bei jedem Tick oder Schedulingereignis hieraus den nächsten auszuführenden Thread ermittelt. Auch hier werden verschiedene Verfahren unterschieden, um die Ausführung von Threads zur Laufzeit zu organisieren (z.B. EDF[7], RMS[8], BE[9], etc.). Aktueller Stand der Technik im Bereich des Echtzeit-Schedulings ist eine Kombination von FCFS- (oder FIFO-) und RR-Verfahren mit Prioritäten in Form eines Multilevelschedulings. Für Echtzeit-Anwendungsteile wird FCFS auf re-

[5] CISC = Complex Instruction Set Computer, 200-400 Maschinenbefehle
[6] RISC = Reduced Instruction Set Computer, 50-150 Maschinenbefehle
[7] EDF = Earlies-Deadline-First - Kürzester wartender Thread wird als nächstes ausgeführt
[8] RMS = Rate-Monotonic-Scheduling - Höchstfrequenter Thread hat höchste Priorität
[9] BE = Best-Effort - Keine besondere Echtzeitfähigkeit, aber schnelle Hardware verwendet

lativ hohem Prioritätsniveau und für Dialog- und Hintergrundbetrieb eher RR auf niedriger Prioritätsebene verwendet. Beispiele hierfür sind Solaris, QNX und Linux. Bei der Verwendung von RR besteht generell die Schwierigkeit, dass kleine Zeitscheiben einen verhältnismäßig großen Overhead erzeugen, während große Zeitscheiben zu langen Wartezeiten führen können. Die Verwendung von FIFO-Scheduling birgt die Notwendigkeit zur Integration spezieller Mechanismen für die Vermeidung von Zykluszeitüberschreitungen, da gerade in höheren Prioritätsebenen sonst keine Kontrolle für das Ende einer Thread-Ausführung existiert.

## 2.2 Systemsoftware-Implementierungen

Mittlerweile existieren mehr als 100 Echtzeit-Betriebssysteme und Echtzeit-Betriebssystemerweiterungen, von denen ca. 20 häufig eingesetzt werden. Da es den Rahmen dieser Arbeit sprengen würde, auf alle Besonderheiten der einzelnen Betriebssysteme einzugehen, wird in diesem Abschnitt die Leistungsfähigkeit einiger weniger gängiger Echtzeitbetriebssysteme dargestellt. Die für Echtzeitanwendungen wesentlichen Leistungsmerkmale können so für die unterschiedlichen Implementierungen miteinander verglichen werden, um eine Entscheidungsgrundlage für die Auswahl einer geeigneten Systemsoftware aufzubauen. Viele Echtzeitbetriebssysteme nutzen dabei die Schnittstellendefinitionen des POSIX[10]-Standard, um dem Benutzer seine Systemsoftwareimplementierung zur Verfügung zu stellen. Die bei der Implementierung verwendete Systemarchitektur entscheidet dabei über die Zuverlässigkeit und Leistungsfähigkeit der Betriebssysteme.

### 2.2.1 Windows

Die heute bekannteste und sehr weit verbreitete Betriebssystem-Familie ist Microsoft Windows. Ihre Entwicklungsgeschichte begann im Jahr 1981 mit der Veröffentlichung von Microsoft DOS 1.0, das damals nur mit Disketten zu betreiben war und einen RAM-Speicher mit der Größe von 8 kByte belegte. Mit dem Fortschreiten der Mikroprozessor-Entwicklung (Intel) wuchs auch die Funktionalität und Benutzerfreundlichkeit. Windows 2000 beispielsweise kommt heute mit 29 Millionen Codezeilen daher. In den folgenden Abschnitten steht nicht der funktionale Unterschied zwischen den unterschiedlichen Standard-Windows-Versionen (NT, 2000, XP, Vista) im Vordergrund; es wird dagegen eine Gegenüberstellung bezüglich harter Echtzeitfähigkeit angestrebt. Aus diesem Grund beschränkt sich die Benennung der Windows-Familie im Folgenden auf „Windows“ ohne Versions-Zusatz.

Abbildung 2.12 zeigt das Blockschaltbild der Systemarchitektur vom heute noch aktuellen Windows 2000, wie sie auch in [13] beschrieben wird. Dieses Betriebssystem trennt, wie praktisch alle anderen Betriebssysteme, anwendungsorientierte Software von den eigentlichen System-Funktionalitäten. Letztere gruppieren sich in der Executive, der Hardware-Abstraktionsschicht, dem Gerätetreiber und dem Mikrokernel,

[10] POSIX = Portable Operating System Interface, Echtzeitbetriebssystem-Standard

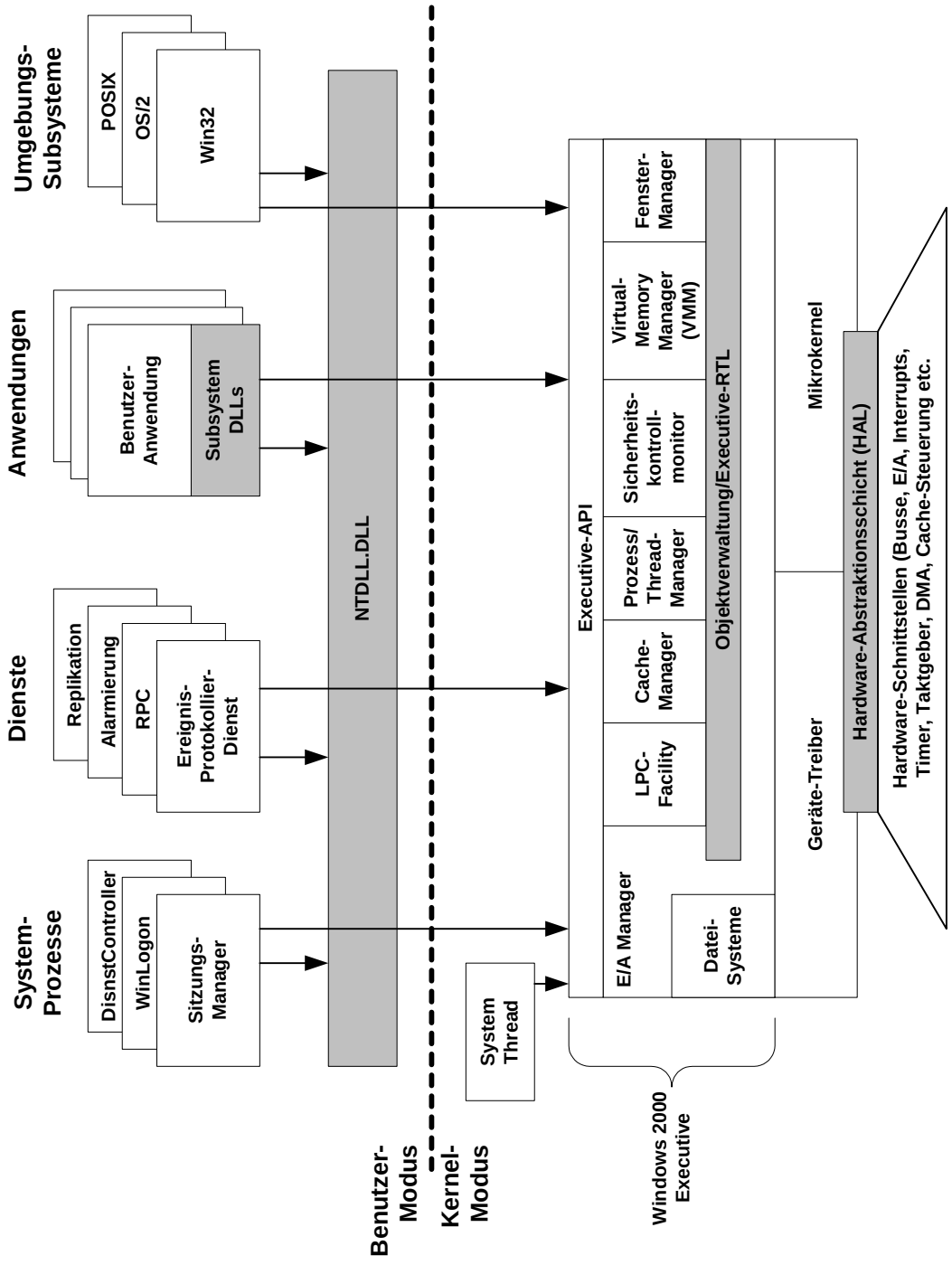



Abbildung 2.12: Windows 2000 Architektur [12]

die durch ihren Ablauf im Kernel-Modus uneingeschränkten Zugriff auf Systemdaten und Hardware haben. Windows verwendet keine reine Mikrokernel-Architektur, sondern eine modifizierte Architektur, welche System-Funktionalitäten zur Optimierung in Funktionsgruppen auslagert, die außerhalb des Mikrokernels liegen. Jede Systemfunktion wird von nur einer Komponente des Betriebssystems verwaltet, auf die über eine Standardschnittstelle zugegriffen werden kann; auf wichtige Systemdaten kann nur über die zugehörige Funktion zugegriffen werden. Auf diese Weise kann jedes Modul ohne Beeinflussung der anderen Module ausgetauscht werden, ohne dass das gesamte System oder verwendete APIs neu geschrieben werden müssen.

Da Windows auf einer Vielzahl anderer Hardware-Plattformen lauffähig sein soll (Übertragbarkeit) und nicht nur auf Intel-Prozessoren, wurde großer Wert auf ein durchgängiges Schichtenmodell gelegt, dessen Aufbau in Anlehnung an Abbildung 2.12 hier kurz erläutert wird [12]:

**Hardware-Abstraktionsschicht (Hardware Abstraction Layer, HAL)** Diese Softwareschicht isoliert den Kernel von Plattform spezifischen Unterschieden hinsichtlich der verwendeten Hardware des Systems. Die Hardware-Abstraktionsschicht übersetzt die physikalischen Ein- und Ausgaben der Hardware (allg. elektrische Signale) in die für die Plattform typischen Befehle. Damit sorgt sie dafür, dass der Zugriff auf eine spezifische Hardware-Funktionalität jedes Rechners (Direct Memory Access (DMA), der Systembus, die Interrupt-Handler, die System-Timer, das Speichermodul etc.) für den Kernel immer gleich aussieht.

**Mikrokernel** Der Mikrokernel umfasst die am häufigsten verwendeten und einfachsten Funktionalitäten der System-Software. Er verwaltet das Scheduling, die Prozess-Umschaltung, die Ausnahme- und Interrupt-Behandlung sowie die Synchronisation mehrerer Prozessoren.

**Gerätetreiber** Diese Schicht enthält das Dateisystem und die Hardware-Gerätetreiber, die die Funktionsaufrufe des Benutzers zur Ein- und Ausgabe in hardwarespezifische Ein- und Ausgabe-Aufträge übersetzt.

Windows ist so konzipiert, dass es auf die Bedürfnisse eines einzelnen Benutzers in einer interaktiven Umgebung oder in der Rolle eines Servers so schnell wie möglich reagiert. Es implementiert einen unterbrechenden Scheduler, bei dem die Prozessorzuteilung nach dem Round-Robin-Prinzip erfolgt. Zusätzlich ändert Windows die Prioritäten der Threads dynamisch, je nach momentanem Thread-Wartezustand und der aktuellen Umgebung. So werden Threads bevorzugt, die auf Ein- oder Ausgabe warten (blockiert) und CPU-lastige Threads (die ihre Round-Robin-Zeitscheibe voll ausfüllen) benachteiligt.

### Real-Time Erweiterungen für Windows

Windows ist nicht in der Lage, harte Echtzeitanforderungen umzusetzen. Allerdings ist durch die weite Verbreitung dieses Betriebssystems, die hochentwickelten Technolo-

gien (Sicherheitsmechanismen, Kommunikationseigenschaften und Visualisierungsfähigkeiten) und den umfangreichen Erfahrungsstand im Umgang mit den zugehörigen ausgereiften Entwicklungstools ein Einsatz in vielen Anwendungsfällen nicht wegzudenken. Sollen diese herausragenden Eigenschaften auch im industriellen Einsatz genutzt werden, müssen jedoch Vorgänge für Kommunikation und Datenaufbereitung (gemäß der Definition von Echtzeit) deterministisch abgearbeitet werden können. Wegen der Bestrebungen, diesen Erfahrungsstand weiter nutzen zu können, wurden und werden ständig neue Echtzeiterweiterungen für Windows entwickelt. Dabei sind im Zusammenhang mit Windows folgende Voraussetzungen zu beachten [14]:

**Schutz vor anderen Anwenderprogrammen im System** Es ist notwendig, dass Regelungs- und Steuerungssysteme mit Echtzeitanforderungen, die neben anderen Windows-Anwendungen ablaufen, nicht von diesen in ihrem zeitlichen Verhalten beeinflusst werden können.

**Schutz vor Windows Systemfehlern** Schlecht implementierte Treiber oder HAL-Erweiterungen können das Gesamtsystem zum Absturz bringen, was zu einem Erscheinen des charakteristischen „blue screen“ führt. Echtzeitanwendungen müssten selbst während und nach einem solchen Absturz unbeeinflusst weiter arbeiten können.

**Überdauern eines Festplattenabsturzes** Mechanische Speichermedien wie Festplatten beinhalten das höchste Fehlerrisiko, da sie auch rotierende und damit alternde Bauteile beinhalten. Echtzeitanwendungen dürfen selbst von solch einem mechanischen Fehler nicht beeinträchtigt werden.

**Robustheit des Echtzeitsystemteils** Viele der heutigen Echtzeitbetriebssysteme haben eine Entwicklungsgeschichte von mehreren Jahren und sind über diese lange Zeit in vielen Anwendungen erprobt und optimiert worden. Je größer und komplexer aber ein Betriebssystem ist, umso schwieriger ist es, innerhalb kurzer Zeit eine in allen Situationen garantiert funktionierende Systemänderung zu implementieren.

Zwei grundsätzliche Herangehensweisen sind praktikabel, um Windows mit harten Echtzeiteigenschaften auszustatten. Zum einen ist es möglich, den HAL (unterste Software-Ebene, die eine allgemeine Schnittstelle zur Hardware definiert und bereithält) zu modifizieren und damit die Verwendung z.B. des Hardware-Timers und der Interrupts neu zu organisieren. Dieses Verfahren erfüllt von seiner Struktur her nur mit Einschränkungen die oben erwähnten Voraussetzungen. Eine andere Möglichkeit besteht darin, ein etabliertes Echtzeitbetriebssystem mit Windows zu kombinieren und auf diese Weise Windows als einen niederprioren Task weiter zu betreiben. Auf diese Weise sind alle oben erwähnten Anforderungen einfacher erfüllbar, da es sich dann um zwei in sich konsistente und erprobte Betriebssysteme handelt, deren Verbindung neu implementiert werden muss.

Eine besonders verbreitete Variante des ersten Ansatzes ist **RTX** von Venturcom [19]. Abbildung 2.13 zeigt den strukturellen Aufbau von RTX in der Windows-Betriebssystem-Umgebung. Das Real-Time-Subsystem verhält sich wie alle anderen Subsysteme

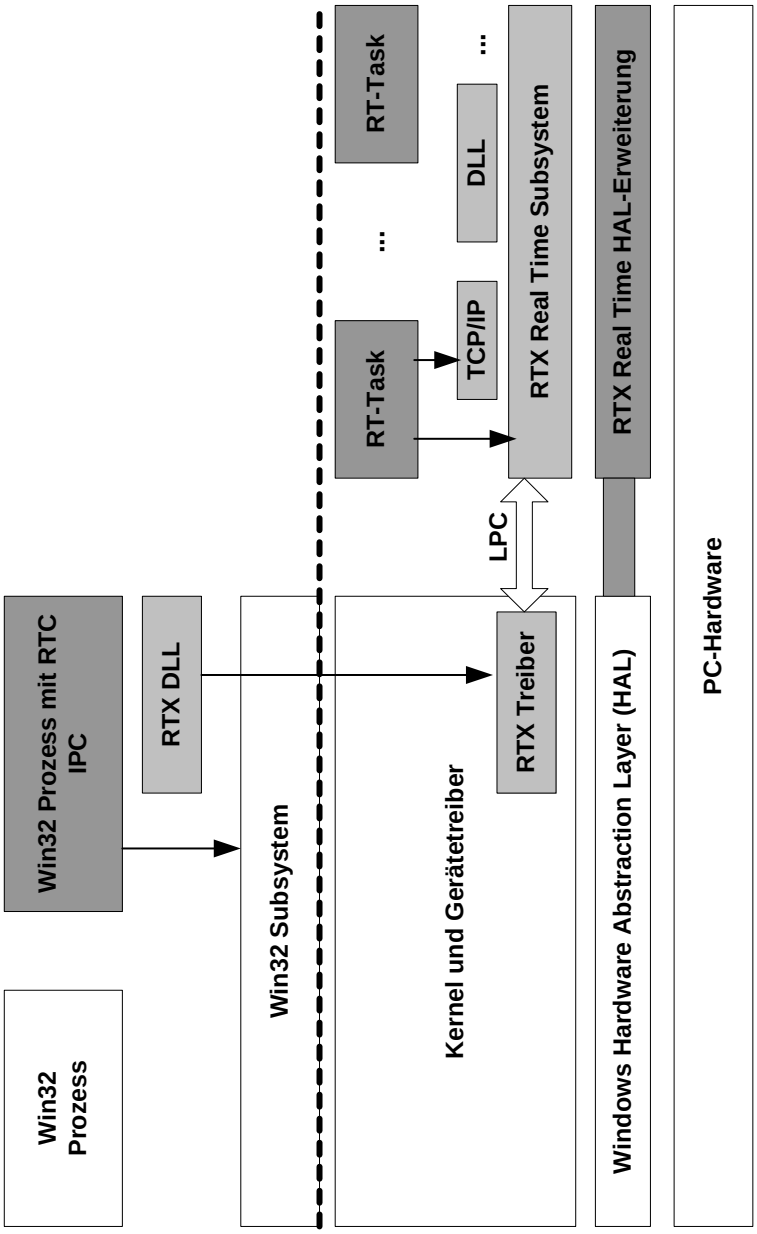



Abbildung 2.13: RTX Echtzeit-Subsystem

(DOS, POSIX oder Win32). Es können bei der Entwicklung von Echtzeitapplikationen also die gewohnten Entwicklungsumgebungen genutzt werden. RTX setzt einen eigenen Thread-Scheduler ein, der dafür sorgt, dass die Prozessorzuteilung für die Echtzeit-Threads immer mit höherer Priorität erfolgt, als das bei den Standard-Windows-Threads der Fall ist. Dazu werden 128 zusätzliche Interrupt-Prioritäts-Stufen oberhalb der 32 vorhandenen bereitgestellt, so dass Windows keine Interrupt-Maskierung der zusätzlichen Quellen durchführen kann. Die RTX-Struktur erlaubt einen direkten Hardwarezugriff und überdauert den charakteristischen „blue screen“.

Realisierungen der zweiten Variante (die Kombination zweier eigenständiger Betriebssysteme) werden für die Kombination von Windows und Windows CE (**CeWin**) in [17] beschrieben, während eine Anwendung der Firma Kuka, in der Windows und VxWorks (**VxWin**) kombiniert wird, in [14] angesprochen wird. Auch für den Embedded-Bereich existieren einige eigenständige, Windows-ähnliche Betriebssysteme, die aufgrund ihres Anwendungsbereichs auf geringen Speicherbedarf optimiert sind. Windows CE wurde nach der Version 3.0 nicht weiter verbessert [15]; an seine Stelle trat mit halbem Speicherbedarf (210 kByte), mehr als fünffacher Interrupt-Reaktionszeit und der Integration neuer Internet-Technologien CE.net [16]. Auch Windows NT/XP wird in einer Embedded-Version angeboten, dessen Echtzeit-Verhalten für Anwendungen im Mess- und Regelungstechnikbereich allerdings nicht ausreichend ist [17]. Aus diesem Grund wird häufig versucht, die mangelnde strukturbedingte Echtzeitfähigkeit durch proprietäre Software-Erweiterungen zu umgehen, so dass mittlerweile zahlreiche Lösungen verfügbar sind.

### 2.2.2 UNIX / Linux

Linux ist ein Betriebssystem, das als UNIX-Variante für die IBM-PC-Architektur ins Leben gerufen wurde. 1991 veröffentlichte Linus Torvalds, ein finnischer Informatikstudent, eine erste Linux-Version im Internet. Dieses Betriebssystem ist samt Quellcode bei der Free Software Foundation (FSF) erhältlich, deren Ziel es bis heute ist, stabile, plattformunabhängige Software bereitzustellen, die kostenlos verfügbar und von hoher Qualität ist sowie gleichzeitig von einer Benutzergemeinschaft getragen wird. Die Größe des Quellcode beschränkt sich im Vergleich zu Windows 2000 auf nur 1 Millionen Zeilen (ohne X). Die FSF betreibt das Projekt GNU, unter dem Tools für Software-Entwickler zur Verfügung gestellt werden und stellt Software als General Public License (GPL) zur Verfügung.

Linux ist ein monolithisches Betriebssystem. Praktisch bedeutet dies, dass die Systemsoftware aus einer Ansammlung von Funktionen besteht, die sich gegenseitig aufrufen können. Der Anwender nutzt eine klar definierte Anwenderschnittstelle, um über Systemaufrufe auf die Betriebsmittel des Rechners (Hardware und Software) zuzugreifen (grau hinterlegte Blöcke in Abbildung 2.14). Realisiert wird diese Trennung zwischen Systemsoftware und Benutzerprogramm über einen Software-Interrupt, der die CPU zwischen dem sogenannten Nutzer-Modus und System-Modus umschalten kann. Um den Nachteil der Unflexibilität eines monolithischen Betriebssystems auszugleichen, wurde Linux um eine Modularchitektur erweitert, die es ermöglicht, Treiber auszulagern bzw. neue Funktionalitäten einfach in die Kernel-Funktionalität

einzugliedern. Trotzdem ist Linux kein präemptives Betriebssystem, da die Kernel-Routinen vom Scheduler nicht unterbrechbar sind [18]. Ein neues Scheduling eines Anwender-Threads erfolgt dagegen nur unter einer der folgenden Bedingungen:

- Wenn die globale Systemvariable `need_resched` beim Beenden einer Systemfunktion auf eins gesetzt ist (durch Interrupt-Routine)
- Wenn der Scheduler im Benutzer-Modus direkt aufgerufen wird (z.B. durch `sleep()`)
- Wenn ein Prozess keine Systemfunktion ausführt und durch einen Uhrentick unterbrochen wird
- Wenn im Kernel-Modus auf ein belegtes Betriebsmittel gewartet werden muss
- Wenn kein Prozess im System rechenbereit ist, ruft der Idle-Prozess den Scheduler zyklisch auf

Nimmt der Scheduler seine Arbeit auf, so ermittelt er den nächsten auszuführenden Thread gemäß einer der drei Scheduling-Mechanismen `SCHED_OTHER`, `SCHED_FIFO` und `SCHED_RR` und startet ihn. Aus der Aufzählung der Bedingungen ist erkennbar, dass Linux gerade mal für Applikationen mit „weichen“ Echtzeit-Anforderungen geeignet ist, da keine verbindlichen Aussagen über Reaktionszeiten des Schedulers gemacht werden können, egal welcher Scheduling-Mechanismus eingestellt ist.

Ab der Version 2.6 ist Linux bedingt geeignet für Echtzeitanwendungen, da über die prinzipiell echtzeitfähigen Schedulingverfahren (RR und FIFO) präemptives Scheduling mit einer Zeitauflösung von nun 1ms (im Vergleich zu bisher 10ms) realisiert wird. Außerdem wurde der Scheduler mit einem Aufwand O(1) realisiert und nicht mehr wie bisher O(N). Trotzdem bleibt der Kernel nach wie vor nicht unterbrechbar, so dass es zu unbestimmten Latenzzeiten kommen kann. Verschiedene Patches gehen diese Defizite zwar an, können aber zusammen keine harten Echtzeitanforderungen erfüllen [20].

### Real-Time Linux

Um Linux für Anwendungen mit „harten“ Echtzeit-Anforderungen zu modifizieren, wurde unter dem Namen Real-Time-Linux ein eigenes Echtzeitbetriebssystem entwickelt, das unter den normalen Linux-Kernel gesetzt wird. Es existieren auch weitere Varianten von Real-Time-Linux, wie z.B. das Real Time Application Interface (RTAI), das ebenfalls die gleiche System-Architektur wie RT-Linux verwendet, aber als Open-Source-Realisierung erhältlich ist. Ein Bestreben war dabei, die Interrupt-Handling-Schicht weiter zu vereinfachen (Real-Time Hardware Abstraction Layer, RTHAL), so dass darauf nahezu jedes beliebige Echtzeit-Betriebssystem aufsetzen könnte. Abbildung 2.14 zeigt die Architektur von RT-Linux. Man erkennt, dass der ursprüngliche Linux-Kernel als RT-Task mit niedrigster Priorität läuft, während weitere RT-Tasks parallel dazu ablaufen können und von einem eigenen RT-Scheduler

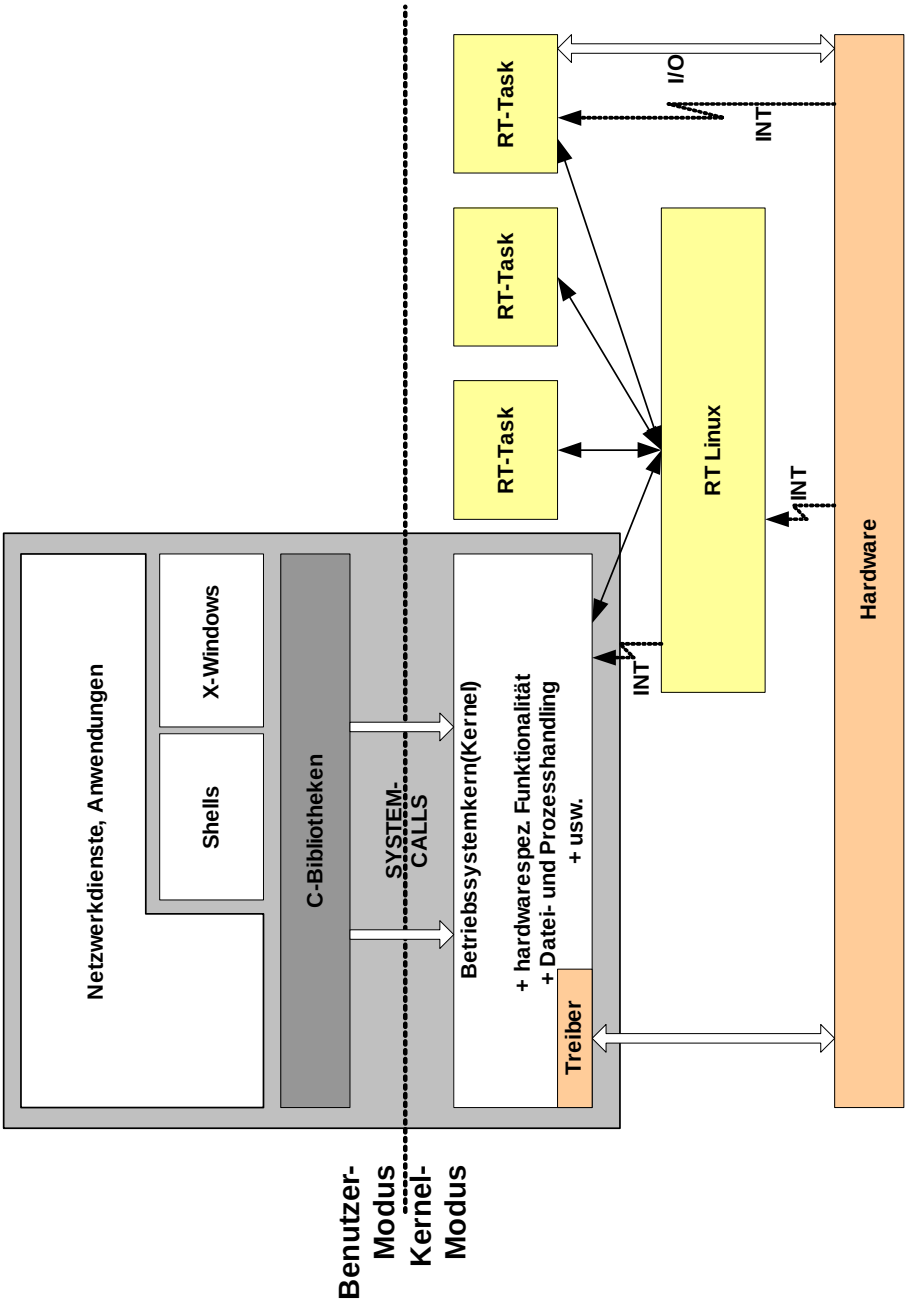



Abbildung 2.14: RT-Linux Architektur

verwaltet werden. Der Linux-Kernel läuft also im Hintergrund und bearbeitet alle bisherigen Anwendungen mit „weichen“ Echtzeit-Anforderungen, während RT-Linux die Anwendungs-Teile übernimmt, die auf eine „harte“ Echtzeitfähigkeit angewiesen sind; diese wird dadurch erreicht, dass die Hardware-Interrupts umgelenkt werden. Die Architektur von RT-Linux teilt sich hierbei auf folgende Module auf:

- RT-Linux Kern Modul (Kernel Patch)
- Scheduling Modul
- Kommunikations-Modul (FIFO, Shared Memory)
- Modul mit Synchronisations- und Interprozess-Kommunikations-Primitiven (optional)
- Modul, durch das ausgewählte Parameter des Echtzeitsystems verändert werden können

Die Programmierschnittstelle von RTLinux ist POSIX-Konform. Sie bietet die Schedulingverfahren EDF, FIFO und „sporadische Prozesse“ an, während eine Prozesssynchronisation über die Verwendung von Semaphoren und Mutexen ermöglicht ist. Die Kommunikation basiert auf Shared-Memory und Pipes. Nachteil dieser Struktur ist, dass sämtliche RT-Applikationen explizit für dieses System geschrieben werden müssen und es nicht möglich ist (Version 1.1), daraus Systemfunktionen des Standard-Linux aufzurufen. Außerdem führt ein Absturz eines RT-Prozesses zum Absturz des gesamten Systems. Der Umstand, dass alle RT-Tasks im gleichen Kernel-Adressraum arbeiten, macht das System fehleranfällig, besonders wenn die RT-Tasks von unterschiedlichen Programmierern implementiert werden.

### 2.2.3 VxWorks

VxWorks wird von der Firma WindRiver Systems entwickelt und vertrieben. Es hat als „reines“ Betriebssystem (im Gegensatz zu den oben erwähnten Betriebssystem-Erweiterungen) einen hohen Bekanntheitsgrad erlangt. Die Anwendung wird stets auf einem Host-System (Windows oder UNIX) entwickelt und wird dann auf ein Target-System geladen. Dabei kommen proprietäre Entwicklungsumgebungen (zunächst über die Anwendung „Tornado“ und später auf der Umgebung „Eclipse“ basierend) mit Simulationsumgebung und integriertem Debugger zum Einsatz. VxWorks ist ein durch Modifikation und Neukompilieren skalierbares Echtzeitbetriebssystem.

Im Gegensatz zu Linux unterscheidet VxWorks keine User- bzw. Protected-Modi. Bis zur Version 5.5 existierte kein Speicherschutz für Prozesse, was sich - wie im Fall von RTLinux - nachteilig für Entwicklungen mit vielen Modulen und unterschiedlichen Entwicklern bemerkbar macht. Die Programmierschnittstelle nutzt eigene sowie POSIX-konforme Funktionen. Unterstützte Synchronisations- und Kommunikationsmechanismen beschränken sich auf Semaphoren, Mutexe, Nachrichten-Warteschlangen und Signale. Als verwendbare, präemptive Schedulingverfahren wer-

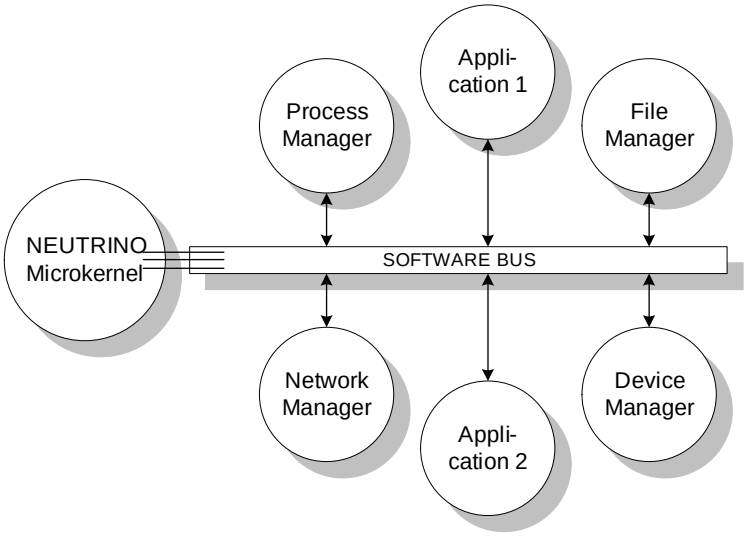



Abbildung 2.15: Microkernelstruktur von QNX Neutrino

den nur RR und FIFO angeboten, die die Ausführung von Prozessen mit Prioritäten zwischen 0 und 255 organisieren. Eine Uhrenauflösung ist dabei bis 30 kHz möglich.

### 2.2.4 QNX Neutrino

Die Firma QNX Systems entwickelt das Echtzeitbetriebssystem QNX mit einem sehr kleinen Kernel (in der Version 4.24 ca 11kB). In diesem sind sehr gut ausgearbeitete und grundlegende Echtzeit-POSIX-Funktionen zur Verwaltung von Threads, Signalen, Scheduling und Timern enthalten. Neben dem Kernel sind weitergehende Dienste als Systemprozesse realisiert, die als Anwenderprozesse behandelt werden. Abbildung 2.15 zeigt die einfache Struktur des Betriebssystems QNX Neutrino. Der wichtigste Prozess ist dabei der Prozess-Manager, der die Schedulingverfahren FIFO, RR, Adaptiv (bei Verbrauch der Zeitscheibe erfolgt eine Erniedrigung der Threadpriorität um eine Prioritätsstufe) und „sporadisches Scheduling“ verwendet, um die Threads mit Prioritäten zwischen 0 und 64 zu verwalten. Ein Skalieren des Betriebssystems erfolgt durch einfaches Laden oder Entfernen von System-Modulen. Die Entwicklung von Applikationen kann direkt auf dem Zielsystem erfolgen und wird durch einen Standard-Editore und neuerdings auch komfortable Entwicklungswerkzeuge (z.B. Eclipse) und Debugger unterstützt.

Die Kommunikation wird vor allem durch einen blockierenden Nachrichtentransfer realisiert. Aber auch Signale, Pipes, FIFOs, Nachrichten-Warteschlangen und Shared-Memory können verwendet werden, um den effektiven Datentransfer in Applikationen zu ermöglichen. Dabei findet zwischen den Prozessen ein Speicherschutz Anwendung, der modulare Softwarekomponenten auch verschiedener Entwickler voneinander trennt. Eine Synchronisation von System- oder Anwendungszuständen kann über Se-

maphore, Mutexe, Konditionsvariablen, Leser/Schreiber-Blockaden und Barriers, die alle entsprechend dem POSIX-Standard zugreifbar sind, geschehen. Die Anwendung von Prioritätsvererbung verhindert, dass es zu Prioritätsinversion kommt.

Wegen der erwähnten Vorzüge wurde zur Realisierung der in dieser Arbeit vorgestellten Kommunikations-Infrastruktur und der gesamten Robotersteuerung das Echtzeitbetriebssystem QNX Neutrino gewählt.

## 2.3 Middleware Lösungen

Modularisierung technischer Systeme bringt Vorteile mit sich. Funktionskomponenten können leicht ausgetauscht und in anderen Anwendungen übernommen werden. Erweiterungen sind leicht realisierbar. So ist es zum Beispiel selbstverständlich, dass die PC-Hardware den Bedürfnissen des Anwenders durch das Einsetzen entsprechender Bus-Karten angepasst werden kann. Auch im Bereich der Software gibt es Bestrebungen zur Modularisierung. Mit der Einführung der objektorientierten Programmierung ist ein wesentlicher Schritt zur Realisierung modularer Software gemacht worden. Fertige Module bzw. Klassen können ausgetauscht werden.

Die Modularität dieses Programmierkonzeptes ist aber noch eingeschränkt. Die Schwierigkeit liegt in der Kommunikation der Softwaremodule untereinander. Es kommt zu einem Netzwerk von Client/Server-Interaktionen. Dieses nimmt häufig komplexe Ausmaße an. Die Systemübersicht ist dadurch eingeschränkt. Ein einfacher Austausch von Softwaremodulen ist nicht ohne weiteres möglich, da dabei in der Regel das gesamte Systemverhalten inklusive der zeitlichen Kommunikationsabläufe berücksichtigt werden muss. Dies kann bei umfangreichen Systemen nicht gewährleistet werden. Daher ist auch die Erweiterung des Systems schwierig: Kommunikationsschnittstellen müssen zuerst geschaffen werden. Dies führt zu Änderungen an den übrigen Modulen. Daher schafft eine bloße Unterteilung eines Systems in unterschiedliche Module noch keine wirkliche Modularität.

Abhilfe schafft hier die Verwendung einer Middleware. Sie hat die Aufgabe, die verfügbaren Server eines Systems zu verwalten, zu organisieren und deren Dienste für Clients bereitzuhalten. Diese können dann über die Middleware Kontakt zum Server aufnehmen. Hierbei werden alle notwendigen Kommunikationsschritte von der Middleware erledigt. Das hat den Vorteil, dass die Serverimplementierung, die verwendete Hardware und die Kommunikationsart für den Client verborgen bleiben. Die Dienste können von ihm über das Kommunikationsinterface der verwendeten Middleware einfach genutzt werden. Der Vorteil liegt auf der Hand: Die installierten Module eines Systems können über ein standardisiertes Interface angesprochen werden.

Abbildung 2.16 zeigt den exemplarischen Aufbau eines einfachen, zentral organisierten Steuerungssystems. Trotz Modularisierung von Funktionalitäten kommt es im Fall (a) zu komplexen Kommunikationsbeziehungen, bei denen das Auslösen oder kommunikationstechnische Modifizieren eines Funktionsmoduls nur unter implementierungstechnischem Aufwand auch innerhalb angrenzender Module zu realisieren wäre. Fall (b) verwendet eine Middleware, über die der gesamte Datenaustausch aller beteilig-

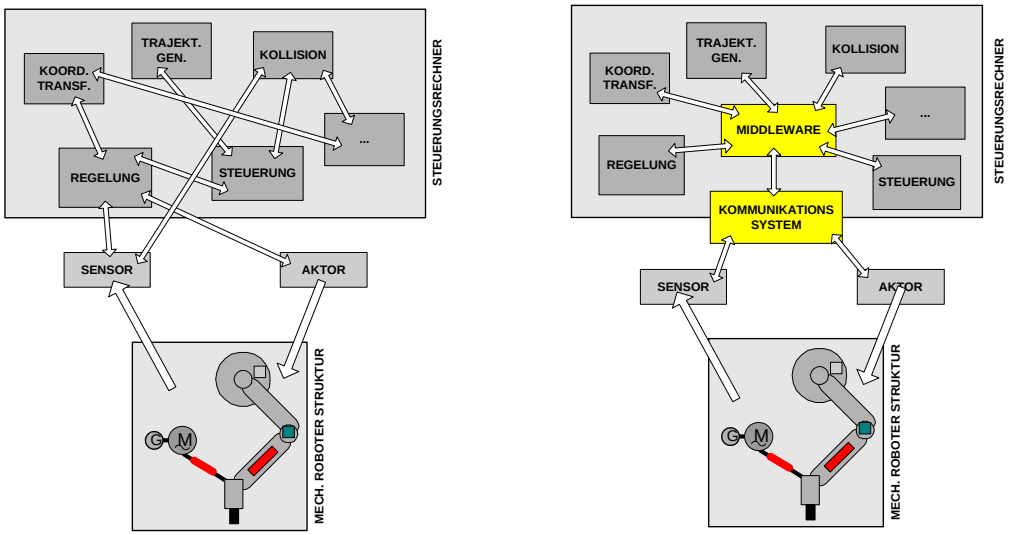



Abbildung 2.16: Beispiel der Modularisierung eines Steuerungssystems ohne (a) und mit (b) Verwendung einer Middleware

ten Module abgewickelt wird. Ein Auslösen oder Modifizieren eines Moduls kann nun selbst zur Laufzeit des Systems problemlos erfolgen, da hier für alle Kommunikationsanfragen nur eine einzige verwaltende Instanz existiert.

Als Middleware wird eine Softwareschicht bezeichnet, die für den Anwender bzw. Programmierer eine Programmierabstraktion auf Basis standardisierter Schnittstellen und Protokolle zur Realisierung transparenter Kommunikation bereitstellt und damit die Heterogenität der zu Grunde liegenden Plattformen (Netzwerke, Hardware, Betriebssysteme und Programmiersprachen) verbirgt [107] [104]. Je nach Anwendungsbereich werden dabei unterschiedliche Mechanismen zur Kommunikation, Kooperation und Synchronisation der Applikationsprozesse angeboten.

Im Allgemeinen werden Middlewarelösungen für ein breites Anwendungsspektrum konzipiert und hierbei angestrebt, einer große Menge von Randbedingungen gerecht zu werden. Dies führt in den meisten Implementierungen zu einem enormen Verwaltungs-Overhead, den sich beispielsweise Echtzeitsysteme, bei denen die Zeitkonstanten in Mikro- bis Millisekundenbereich liegen, in der Regel nicht leisten können. Zudem führt der Overhead oft zu einer aufwändigen Handhabung der Middleware, was den praktischen Einsatz erschwert.

Heute existieren bereits viele unterschiedliche Middlewaresysteme, die sich im Bezug auf Anwendungsbereich, Betriebssystemunterstützung, Funktionsumfang, Programmier-/Benutzerschnittstelle, Leistungsfähigkeit u.v.m. unterscheiden. Es lassen sich eine Reihe von generellen Middlewareklassen unterschieden, deren Einsatz jeweils für verschiedene Anwendungsgebiete passend ist [5]:

**dokumentenbasiert** Die Idee ist es, ein verteiltes System wie eine durch Hyperlinks verbundene Ansammlung von Dokumenten aussehen zu lassen. Über die Spezi-

fikation eines Zugriffsprotokolls, eines Servernamens und des gewünschten Dateinamens kann der Nutzer auf verteilt vorliegende Informationen zugreifen, sofern er weiß, dass sie existieren. Bekanntestes Beispiel für diese Klasse von Middleware ist das *WWW*.

**dateisystembasiert** Das System wird als globales Dateisystem angesehen. Ein Datentransfermodell entscheidet, in welcher Weise auf Dateien zugegriffen werden kann und die Wahl einer zugehörigen Verzeichnishierarchie bestimmt die dazu notwendigen Zugriffswege. Beispiele für solche Middlewaresysteme sind *AFS* und *NFS*.

**objektbasiert** Das verteilte System besteht aus Laufzeitobjekten, auf die über die Nutzung von Objektmethoden nach dem Client/Server-Prinzip zugegriffen werden kann. Für begrenzte Systeme kommt hier oft *CORBA*, für weltweite dagegen z.B. *Globe* zum Einsatz.

**koordinationsbasiert** Es werden keine verteilten Objekte mit Methoden und Attributen verwendet, sondern eine gemeinsame Datenbasis, auf die verteilte Prozesse zugreifen. Die Prozesse (Informationserzeuger oder -verbraucher) werden vollständig voneinander entkoppelt und können über atomare Zugriffe synchronisiert werden. Vorreiter dieser Art von Middleware ist *Linda*, auf dem das bekanntere *Publish/Subscribe* und das *Jini* beruhen.

Bei der Entwicklung von Applikationen begegnen dem Entwickler neben der eigentlichen Softwarefunktionalität auch allgemeine Herausforderungen, die sich z.B. aus der *Heterogenität* der unterlagerten Soft- und Hardwareebenen, der gewünschten *Offenheit* bezüglich Erweiterungen oder Modifikationen, der *Sicherheit* für verwaltete Informations-Ressourcen, der *Skalierbarkeit* der Anwendung, der *Fehlerverarbeitung* bei Zwischenfällen, der *Nebenläufigkeit* in Zusammenhang mit anderen Anwendungen und der *Transparenz* beim Zugriff auf unabhängige Komponenten ergeben. Beim Einsatz innerhalb eines verteilten Systems mit möglicherweise mehreren gleichzeitig arbeitenden Entwicklern sind solche Überlegungen besonders wichtig und können über die Nutzung eines geeigneten Middleware-Ansatzes berücksichtigt werden.

Die Entwicklung von Echtzeitsystemen kann heute weitestgehend über die Nutzung ausgereifter Modellierungswerkzeuge (z.B. Rhapsody [154]) unterstützt werden [155]. Allerdings erfolgt die Entwicklung aus Gründen der Hardwarenähe, der erreichbaren Ausführungsgeschwindigkeit und des Bekanntheitsgrades heute oft noch direkt in den Programmiersprachen C/C++ [108]. Diese sind allerdings programmiertechnisch fehleranfällig, da die Speicherverwaltung, Echtzeit-Ablaufsteuerung und die Verwendung von Datentypen manuell erfolgen muss, was dem durchschnittlichen Programmierer erst nach langer Programmierroutine keine Schwierigkeiten mehr bereitet. Die Vorteile einer Middleware können so auch im Bereich einer praktischen Implementierungs-Unterstützung gesehen werden, da der programmiertechnische Handlungsbedarf bei der Implementierung damit weitestgehend auf funktionale Anteile reduziert werden kann.

In den folgenden Absätzen werden ausgewählte Technologien der heute eingesetzten Middlewarelösungen exemplarisch vorgestellt. Die Idee solcher kommunikationsver-

waltender Systeme ist generell nicht neu; es gibt mittlerweile eine Vielzahl unterschiedlichster Realisierungen für alle möglichen Anwendungsbereiche und Leistungsklassen. Die Anwendung von Middleware im Bereich der Antriebssteuerungen stellt allerdings erhöhte Leistungsanforderungen, die von den meisten Systemen nicht erfüllt werden können. Ausführliche Darstellungen und Bewertungen zugehöriger Softwaretechnologien (COM, DCOM, OLE, OPC, .NET) sind beispielsweise in [115] und [21] vorhanden und werden aus diesem Grund hier ausgelassen. An dieser Stelle erfolgt die Vorstellung der für den Anwendungsbereich Robotersteuerung prinzipiell nutzbaren und auch tatsächlich genutzten Systeme. Dabei liegt der Schwerpunkt auf angebotenen Mechanismen, die die Echtzeitfähigkeit einer Anwendung sicherstellen können. Der Begriff „Echtzeit“ muss sich dabei immer auch auf den Applikationskontext beziehen, womit dann auch das verwendete Rechner- und Betriebssystem sowie die Zielapplikation, mit seinen zeitlichen Randbedingungen, als Einflussgrößen einbezogen werden müssen.

### 2.3.1 CORBA

Im Bereich der objektorientierten Middlewaretechnologien ist und bleibt die *Common Object Request Broker Architechture*, CORBA, die am weitesten verbreitete Spezifikation.

CORBA ist eine Spezifikation der OMG[11] für ein methodenorientiertes Middlewaresystem. Der Kerngedanke ist dabei die Verwendung einer Schnittstellen-Beschreibungssprache, IDL[12], um den Zugriff auf Anwendungsobjekte orts-, sprach- und plattformtransparent zu halten. Die Implementierung der anwendungsspezifischen Objekte obliegt dabei vollständig dem Anwender; die Realisierung der spezifizierten CORBA-Dienste ist Teil der jeweiligen CORBA-Realisierung.

Seit der Version 1.0 (1991) bis zur Version 2.4 (2000) war die Spezifikation ungeeignet für Echtzeitanwendungen. Mit der Version 2.5 (2001) und der Erweiterung auf RT[13]-CORBA wurden auch Funktionalitäten für Einsatzgebiete mit harter und weicher Echtzeitanforderung spezifiziert, die vor allem die Dienstgruppen Prozessorressourcen-Verwaltung und Kommunikations-Verwaltung ergänzten [110]. Auf ihrer Basis wurden zahlreiche Middlewaresysteme implementiert, von denen TAO[14] die am weitesten verbreitete ist [105] [106]. Die TAO-Implementierung dient auch als Grundlage für erweiterte Middleware-Realisierungen wie z.B. TENA [116] und Miro [117].

CORBA-Implementierungen werden immer als „ORB“ (Object Request Broker) bezeichnet und beinhalten drei wesentliche Funktionseinheiten: ORB, Proxy und Skeleton. Abbildung 2.17 zeigt den schematischen Aufbau eines ORB. Der **Proxy** nimmt Clientanfragen lokal entgegen und wandelt sie in ein allgemeines Datenformat um, in dem sie dann weiter übertragen werden (Marshalling). Der **ORB** übernimmt die Netzwerk- und Interprozesskommunikation des Aufrufs. Er ist dabei in jedem betei-

[11] OMG = Object Management Group
[12] IDL = Interface Description Language
[13] RT = Real-Time
[14] TAO = The ACE Orb, implementiert von der Distributed Object Computing Group

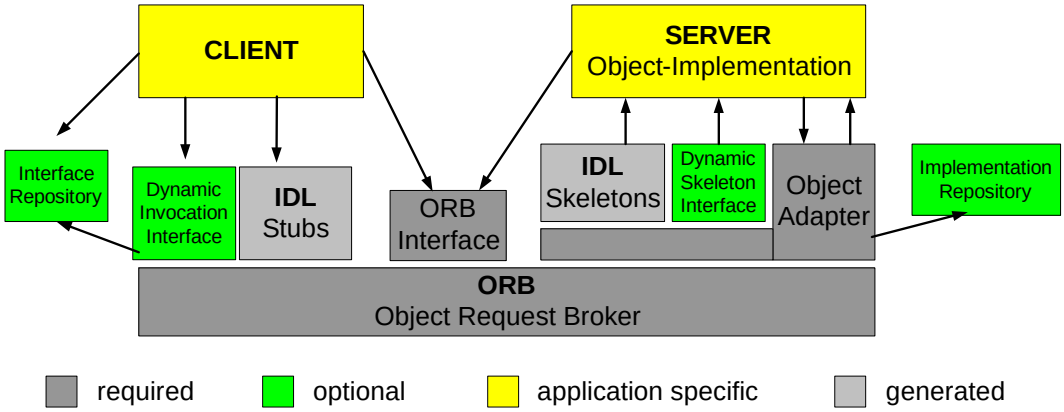



Abbildung 2.17: Aufbau einer CORBA-Implementierung

ligten Prozess auf jedem Zielsystem als eine eigenständige Instanz vorhanden und verwaltet CORBA-Objekte im jeweiligen Prozess. Zur Kommunikation wird eine allgemein gehaltene Protokollspezifikation GIOP[15] verwendet, die sich auf jedem Transportmedium implementieren lässt; für TCP ist es beispielsweise das IIOP[16] oder ein echtzeitfähiges RIOP[17]. Der **Skeleton** übernimmt die Clientanfrage im Zielsystem und bereitet sie für die Serverfunktionalität auf (Demarshalling).

CORBA ermöglicht die Verwendung verschiedener Mechanismen, mit der sich eine Echtzeitfunktionalität für Kommunikationsaufgaben erfüllen lässt. Die Kommunikation basiert auf der sogenannten Synchronous Methode Invocation (SMI), bei der ein aufrufender Client-Thread nach dem Absetzen seiner Anfrage bis zum Empfang der Antwort blockiert bleibt; die Asynchronous Methode Invocation (AMI) lässt den Client dagegen weiter arbeiten und realisiert den Antwort-Empfang über eine Callback-Funktionalität in einem separaten Thread. Obwohl der Datendurchsatz bei Verwendung von AMI gesteigert werden kann, geschieht dies auf Kosten einer erhöhten Systemkomplexität und eines erhöhten Jitters für einzelne Methodenaufrufe [109]. Ein Event-Dienst ermöglicht die Realisierung von Publish-Subscribe-Kommunikation, bei der sich mehrere Module bei anwendungsspezifischen Ereignissen benachrichtigen (aufwecken) lassen können. Diese Funktionalität ist besonders zur Realisierung von Echtzeit-Anwendungen geeignet.

In der Literatur findet man inzwischen Ansätze, die oben genannte Middlewarelösungen im Bereich der Steuerungstechnik einzusetzen. In [89] wird z.B. ein Telerobotik-System basierend auf CORBA vorgestellt. Diese Anwendungen müssen aber keinen harten Echtzeitanforderungen genügen. Im Bereich der Realzeitsoftware mit Zykluszeiten von einigen Mikro- bis wenigen Millisekunden haben Middlewarelösungen bis heute keine praktische Anwendung gefunden. Hier werden nach wie vor die Kommunikationswege zwischen den Softwaremodulen direkt festgelegt. Der Grund hierfür liegt in der Gefahr, die zeitlichen Randbedingungen zu verletzen. Bei festen Kommunikationsstrukturen ist die QoS (Quality of Service) leichter zu garantieren. Dies führt aber wie oben beschrieben zu unflexibler Realzeitsoftware.

[15] GIOP = General Inter Orb Protocol
[16] IIOP = Internet Inter Orb Protocol
[17] RIOP = Real-Time Inter Orb Protocol

Obwohl RT-CORBA auch Mechanismen spezifiziert, die einen Determinismus für Kommunikationsvorgänge sicherstellen sollen, sind die vorhandenen Implementierungen (z.B. TAO) aufgrund ihres Funktionsumfangs und ihrer Generalität nicht als Grundlage für komplexe Robotersteuerungen mit Zykluszeiten unterhalb einer Millisekunde geeignet [111]. Besonders, wenn es darum geht, die realisierte Anwendungsstruktur nicht statisch festzulegen, sondern je nach Umgebungsbedingung dynamisch anzupassen, verhindern die noch deutlich langsameren dynamischen Konfigurationsmechanismen den sinnvollen Einsatz von CORBA-Mechanismen [112].

### 2.3.2 MiRPA

Aus dem Bedarf nach einer schnellen, effektiven Middleware speziell für den Automatisierungs- und Robotikbereich für Montageanwendungen, entstand am Institut für Robotik und Prozessinformatik (iRP) der TU Braunschweig die Implementierung der Middleware MiRPA, die auf die Realisierung von transparenten, netzwerkübergreifenden, nachrichtenbasierten Kommunikationsmechanismen beruht [115] [123]. MiRPA wird am iRP eingesetzt, um als Softwarebus die einzige Kommunikationsschnittstelle zwischen allen im jeweiligen System ablaufenden Anwendungsmodulen darzustellen.

#### Eigenschaften von MiRPA (iRP)

Zwecks Abgrenzung der in dieser Arbeit vorgestellten Kommunikationsinfrastruktur (s. Kapitel 4.2) seien hier die wichtigsten Eigenschaften des am iRP realisierten MiRPA-Ansatzes vorgestellt, die als Entscheidungsgrundlage für bzw. gegen eine Nutzung im SFB 562 dienten. Es wird am Ende des Abschnitts eine Bewertung der Eigenschaften hinsichtlich der SFB-Anforderungen vorgenommen. Mittlerweile ist weiter an dem Mirpa-Ansatz und an seinen Schnittstellen gearbeitet worden, wobei allerdings die grundsätzliche Struktur beibehalten wurde.

**Umsetzung des Client/Server Kommunikations-Modells** Die Realisierung dieses Modells ist grundlegend mit dem nachrichtenbasierten Kommunikationsmechanismus verbunden. Er ermöglicht das klare Abgrenzen von Funktionalitäten, die dann über beliebig viele Clients in Anspruch genommen werden können. Leider ist dieser Mechanismus nicht konsequent implementiert, so dass es Servern gestattet ist, selbständig und spontan Anfragen an andere Server zu versenden; damit kann es z.B. zu Verzögerungen kommen, wenn ein Client auf einen der Dienste des Servers zugreifen möchte. Außerdem besteht die Gefahr von zyklischen Blockaden, die wiederum durch zusätzliche Mechanismen verhindert werden müssen. Die Verwendung der „asynchronen“ Variante führt in ungünstigen Fällen zu großem Nachrichtenaufkommen durch Polling.

**Umsetzung des Publish/Subscribe Kommunikationsmodells** Mehreren unterschiedlichen Servern wird es ermöglicht, gleiche Anfragen zu registrieren und im Betrieb zu bearbeiten. Die gesamte Ausführungszeit zum Übertragen einzelner Nachrichten in den Speicherbereich der jeweiligen Subscriberapplikation ist

stark von den Parametern Serveranzahl und Datenmenge abhängig und daher in dieser Form hinsichtlich der Echtzeitfähigkeit kritisch zu bewerten.

**Umsetzung des Online-Aufbaus von Kommunikationsbeziehungen** Clients und Server können zu jeder beliebigen Zeit neue Nachrichten anmelden oder nicht mehr benötigte abmelden. Während des Verfahrens der Roboterstruktur ist eine solche Möglichkeit allerdings nicht unbedingt förderlich, da der Zeitbedarf für nachrichtenbasierte Konfigurationsänderungen nicht festgelegt ist und somit der reibungslose Kommunikationsablauf kritischer Nachrichten beeinträchtigt werden könnte. Eine Lösung wird in Abschnitt 4.2.1 vorgestellt.

**Umsetzung eines Echtzeit-Namensdienstes** Um für Anfragen und Kommandos den richtigen Server adressieren zu können, wird ein Mechanismus benötigt, der möglichst schnell und mit konstanter Latenz das Vorhandensein und den Aufenthaltsort des Servers gemäß des Nachrichten-Namens ermittelt. Natürlich sind dabei nach Zeichenketten suchende Mechanismen ausgeschlossen. Der verwendete Ansatz über eine Hash-Funktion ist sehr gut für Echtzeit-Anwendungen geeignet.

**Latenzzeiten und Jittern bei der Nachrichtenübertragung** Da das Versenden von Nachrichten nicht augenblicklich erfolgen kann, ergeben sich Latenzzeiten, die vom gewählten Betriebssystem, der Rechengeschwindigkeit und der Implementierung der Kommunikationsfunktionalität abhängen. Für diese Latenzzeiten wird bei MiRPA eine zeitliche Grenze angegeben, die „viel kleiner“ als $150\mu s$ sein soll. Diese Latenzzeit unterliegt weiterhin einer Schwankungsbreite (Jitter) von einigen Mikrosekunden. Für die Anwendung im SFB 562 mit seinen geplanten Zykluszeiten im Bereich von $125\mu s$ ist dies allerdings inakzeptabel, sofern dieser Nachrichtenübertragungsmechanismus bei MiRPA die einzige Form der Datenübertragung darstellt.

**Vermeidung von Deadlocks bei der Nachrichtenübertragung** Da das Client / Server Modell bei MiRPA nicht konsequent umgesetzt ist, ist eine dynamische Deadlock-Überprüfung zur Laufzeit des Roboters notwendig. Solch eine Überprüfung und die damit verbundenen Kommunikationsvorgänge gefährden den reibungslosen Berechnungsablauf innerhalb der Robotersteuerung. Es bleibt wünschenswert, dass Deadlocks bereits beim Systemdesign ausgeschlossen werden.

Trotz der vielen Vorteile wie Modulaustausch, Modulerweiterung, Broadcast und gezielter Redundanz lässt sich bei rein nachrichtenbasierter Implementierung feststellen, dass die „Echtzeitfähigkeit“ bei steigender Systemkomplexität, wie sie beispielsweise in den Anwendungen des SFB 562 vorherrscht, nicht sichergestellt werden kann. Der erzielte maximale Datendurchsatz von 4 MByte pro Sekunde, die großen Latenzzeiten bei der Nachrichtenübertragung - je nach Belastung im Bereich von 10 bis $100\mu s$ - und der fehlende zeitliche Determinismus für zyklische Echtzeitfunktionalitäten spielen dabei eine wesentliche Rolle. Aus den vorausgehenden Ausführungen ist also ersichtlich, dass MiRPA für einen Einsatz im SFB562 nicht geeignet ist. Aus diesem Grund wurde in der vorliegenden Arbeit zur Realisierung einer Middleware ein eigener Ansatz gewählt, der den gestellten Anforderungen genügt. Die Bezeichnung „MiRPA-X“ soll

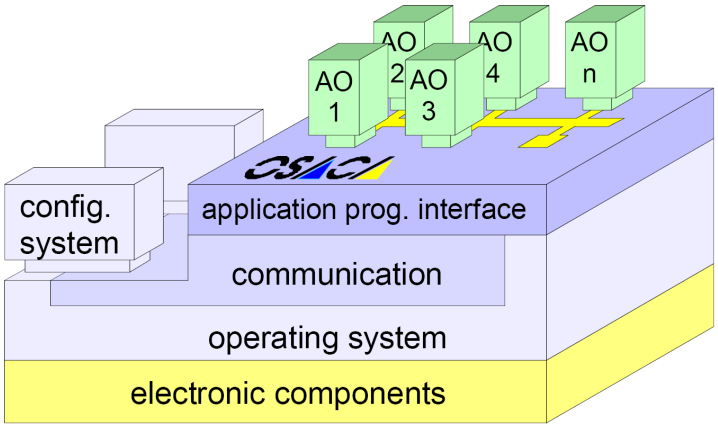



Abbildung 2.18: Struktur eines Steuerungssystems nach OSACA

dabei ausdrücken, dass zwar die Grundidee der Modularisierung und Transparenz mit der von MiRPA übereinstimmt, allerdings das Konzept und die Realisierung der Mechanismen hingegen unterschiedlich sind. Die Realisierung von MiRPA-X wird in Kapitel 4.2 beschrieben.

## 2.4 Steuerungsarchitekturen

Reine Middlewaresysteme füllen den Raum zwischen den Betriebssystemen und der Anwendungssoftware aus und sorgen dabei für eine einheitliche Zugriffsmöglichkeit auf systemweit veröffentlichte Daten und Methoden. In der Regel sind die Anwendungsmodule dabei gleichberechtigt, so dass übergreifende Verwaltungsfunktionen, die für alle Anwendungsmodule bindend sind, nur schwer realisiert werden können, um beispielsweise ein definiertes Scheduling oder zumindest eine garantierte Ablaufsteuerung für Teile der Anwendung zur Verfügung zu stellen. Steuerungsarchitekturen versuchen durch ein strikteres Rahmenwerk unterschiedliche Schnittstellen für spezifische Funktionalitäten zur Verfügung zu stellen. Systementwickler können so spezifische Anwendungsteile (Module) mit einer ihrer Funktion entsprechenden Schnittstelle verbinden. Es entsteht ein Steuerungskonzept, bei dem die Anwendungsmodule je nach verwendeter Schnittstelle unterschiedlichen Softwareschichten zugeordnet werden können.

### 2.4.1 OSACA

Im Allgemeinen stehen Steuerungen für Maschinen als proprietäre Lösungen zur Verfügung. Ihre Soft- und Hardwarekomponenten lassen sich nicht von unterschiedlichen Herstellern beziehen und auch der Zugriff des Bedieners oder weiterer Steuerungssysteme ist nicht einheitlich. Zu diesem Zweck wurde das Projekt *Open Systems Architecture for Controls within Automation Systems*, OSACA, mit dem Ziel der Entwicklung einer Spezifikation für offene und modulare Steuerungsplattformen ins Leben gerufen [119]. Mehrere Vertreter industrieller Steuerungs- und Werkzeugmaschinen-

hersteller sowie öffentliche Forschungseinrichtungen beteiligten sich an diesem Projekt mit dem Ziel der Spezifikation einer herstellerneutralen Referenzarchitektur für offene Steuerungssysteme aller Art (NC[18], RC[19], PLC[20] und CC[21]). Die Spezifikation soll auch offen für neuartige Funktionalitäten und die Verwendung neuer Rechnersysteme sein und erstreckt sich über die Definition von Funktionalitäten (Funktionseinheiten) und den Zugriff auf sie mittels einer plattformunabhängigen Programmierschnittstelle (API).

Abbildung 2.18 zeigt die Struktur eines Steuerungssystems gemäß der OSACASpezifikation. Die von Komponentenherstellern zu entwickelnden Softwareanteile sind die *Architecture Objects* (AO). Dabei werden jeweils eine oder mehrere Funktionsgruppen über ein oder mehrere AOs implementiert. Sie verwenden das OSACA-Programmierinterface, um einheitlich auf unterlagerte Kommunikations- und Betriebssystemschichten zuzugreifen. Die Implementierung an sich ist dabei versteckt; die Daten- und Methodenschnittstelle der AOs sind in ihrer Form spezifiziert und sorgen für Hersteller-Unabhängigkeit. Funktionseinheiten zur Systemkonfiguration, zur grafischen Darstellung und zur Datenbanknutzung sind einheitlich realisiert und sorgen für ein durchgängiges „Look and Feel“.

Um die erwähnten hochgesteckten Ziele zu verfolgen, wurden die Anforderungen an damalig neue Generationen von Steuerungen analysiert (1991). Sie bezogen sich entsprechend der damals vorherrschenden heterogenen Anbietersituation auf die Stärkung von Portabilität für Betriebssysteme, Interoperabilität zwischen den einzelnen Herstellern und Konsistenz hinsichtlich der Schnittstellen zu Nutzern und Automatisierungssystemen. Das Ergebnis ist ein Steuerungskonzept basierend auf einer Middleware; allerdings beschränkt sich deren Anwendung auf die „High- Level“-Funktionalitäten einer Steuerung [120]. Die Berücksichtigung von Echtzeitanforderungen, wie sie besonders für die Anwendung im Bereich der Bewegungssteuerung notwendig sind, ist in der Literatur nicht erwähnt [119] [118]. Der Systemaufbau ist nach dem Bootvorgang statisch festgelegt und bewirkt damit eine hinsichtlich der Anwendung im SFB562 ungünstige Unflexibilität. Die Kommunikationsplattform ist für eine praktische Anwendung unzulänglich dokumentiert und auch durch neuere Entwicklungen bereits überholt. Auch die Steuerungskomponenten sind in Anlehnung an heute veraltete Steuerungssysteme spezifiziert und damit nicht brauchbar für die Arbeiten im SFB 562. Es existieren weder ein industrieller Einsatz noch finden sich Referenzfallstudien.

### 2.4.2 MCA

Die modulare Steuerungsarchitektur MCA (Modular Controller Architecture) ist ein auf der Verwendung von C/C++ basierendes netzwerktransparentes und echtzeitfähiges Rahmenwerk zur Steuerung von Systemen [121]. Es wird vor allem zur Steuerung mobiler Roboter in vielen Projekten des Forschungszentrum Informatik (FZI)

[18] NC = numeric control
[19] RC = robot control
[20] PLC = programmable logic control
[21] CC = cell control

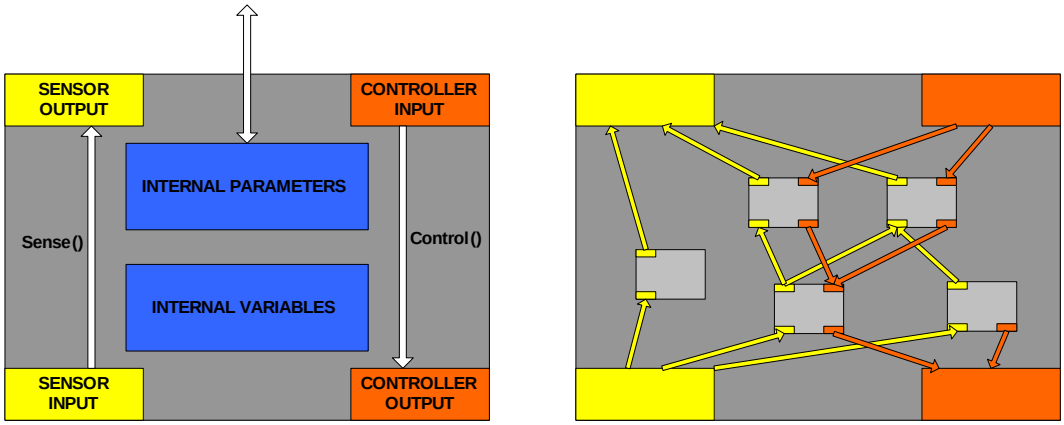



Abbildung 2.19: Grundelemente eines Steuerungssystems nach MCA2

in Karlsruhe [122], der AG Robotersysteme der Uni Kaiserslautern, des Fachbereichs Mechatronik der Uni Duisburg-Essen und des Fraunhofer-Instituts AIS verwendet. Während die ursprüngliche Version von MCA stark an die verwendete Hardware gebunden war und prinzipiell nur zur Unterstützung dieser Hardware-Architektur verwendet wurde, ist MCA2 davon völlig gelöst.

Durch die Definition und Verwendung von wiederverwendbaren Softwaremodulen mit standardisierten Schnittstellen lassen sich mit Hilfe von MCA2 erweiterbare Architekturen realisieren, deren Module übertragbar sind. Abbildung 2.19 zeigt ein Modul (a) und ein zusammengesetztes Modul (b), die die Grundbausteine für eine Robotersteuerung mit MCA2 darstellen. Alle Module sind über 5 Schnittstellen beschrieben: Vier davon verbinden das Modul mit anderen Modulen, welche ober- oder unterhalb angeordnet sein können und jeweils die Ein- und Ausgänge für einzelne Fließkommazahlen oder -arrays darstellen; über eine Schnittstelle können die internen Parameter des Moduls verändert werden. Jedes Modul enthält zwei Funktionen: `Sense()` und `Control()`, die z. B. typischerweise die Berechnung des DKP und des IKP implementieren. Module lassen sich zu hierarchisch organisierten Gruppen zusammenfassen, in denen beim Aufruf der `Sense()`-Funktion alle Sensorpfade von unten nach oben ausgeführt werden; beim Aufruf von `Control()` werden die entsprechenden Pfade der Gruppe von oben nach unten ausgeführt. Die Arbeit der Entwickler beschränkt sich im Optimalfall auf die Entwicklung neuer Funktionen, die bisher noch nicht implementiert wurden. Die Kommunikation zwischen den Komponenten und auch die Synchronisation werden vom Rahmenwerk übernommen, so dass sich die Entwickler auf die Roboterfunktionalitäten konzentrieren können. Die Plattform für MCA2 ist Linux/RTLinux, es wird aber auch Windows unterstützt.

Durch die Verwendung von Modulen mit Standard-Schnittstellen kann im Entwicklungsprozess ein einfaches Testen und Verifizieren der einzelnen Funktionalitäten erfolgen. MCA2 stellt eine grafische Benutzeroberfläche (MCAGUI) zur Verfügung, mit der über TCP/IP übertragene Anwendungsdaten auf einem separaten Rechner visualisiert und parametriert werden können. Die Kommunikation der aufgebauten Module, Gruppen und Komponenten (Gruppen mit festgelegten zeitlichen Ausführungsanforderungen) erfolgt nach außen als „interpart connections“ ebenfalls über TCP/IP-Ver-

bindungen. Dadurch lassen sich zwar Vision-Systeme und externe Roboterkomponenten anschließen, allerdings kann dann keine Echtzeitfähigkeit mehr garantiert werden. MCA2 eignet sich daher nur für autonome (nicht verteilte) Systeme mit Echtzeitanforderungen.

# 3 Ausgewählte Kommunikationssysteme

Das Gebiet der industriellen Kommunikation ist gekennzeichnet durch eine Vielzahl unterschiedlicher Systeme. Durchgängig wird bei der Anwendung eine hierarchische Kommunikationsstruktur verwendet, die aus mehreren Schichten zusammengesetzt ist und zur Beschreibung im Allgemeinen an dem ISO/OSI Referenzmodell angelehnt werden kann. Jede dieser Schichten weist in sich unterschiedliche Anforderungen und Funktionalitäten auf. Jedes der verfügbaren Kommunikationssysteme besitzt dadurch charakteristische Eigenschaften, zum Beispiel bezüglich möglicher Reaktions- und Synchronisationszeiten, der Art und Menge übertragbarer Daten und der Anforderungen an das physikalische Medium. Als Feldbusse werden digitale Kommunikationssysteme bezeichnet, die im Allgemeinen asymmetrisch ausgelegte Geräte (Steuerungen, Sensoren, Stellglieder und Antriebe unterschiedlicher Art) innerhalb der Feldebene einer Automatisierungsanlage miteinander verbinden. Ihre Eigenschaften sind durch die Anforderungen aus dem Einsatz im industriellen Umfeld geprägt. Im Gegensatz zu Anwendungen im Bereich Multimedia bedeutet dies oftmals eine strengere und robustere Definition von Anschlusstechnik, EMV, Determinismus und Synchronität bei der Datenübertragung. Diese Eigenschaften werden auch bei gleichem physikalischem Medium über die Definition und Verwendung spezieller Kommunikationsprotokolle unterschieden und sichergestellt.

Theoretische Grundlagen bezüglich des ISO/OSI-Schichtenmodells und dessen generelle Verwendung in Kommunikationssystemen sind z.B. in [30] beschrieben. In [21] wurden Anwendungsbereiche für unterschiedliche Feldbusse aufgeführt sowie die Kommunikationssysteme SERCOS, MACRO, USB, Fibre Channel, Ethernet und FireWire beschrieben und hinsichtlich ihrer Eignung zum Einsatz in der industriellen Automation bewertet. Die dort dargestellten Ausführungen und Ergebnisse werden innerhalb dieser Arbeit als bekannt vorausgesetzt, so dass an dieser Stelle eine weiterführende Aktualisierung der relevanten Themengebiete erfolgt. Für das Verständnis wesentliche Informationen werden an geeigneter Stelle kurz zusammengefasst dargestellt.

## 3.1 Anforderungen an Feldbusse

Die qualitativen Anforderungen an Feldbusse haben sich seit Beginn der Entwicklung und wachsenden Verbreitung der Feldbusse nicht verändert. Allerdings verschärften sich die quantitativen Anforderungen an Feldbusse aufgrund neuer und komplexerer Anwendungsgebiete der letzten Jahre, besonders im Bereich der Roboteranwendun-

gen. Die folgende Liste stellt einige Anforderungen an Feldbussysteme zusammen, die zu ihrer Bewertung herangezogen werden können [3] [26]:

- Architekturmodell für verteilte Automatisierung
- leistungsfähige Geräteprofile
- Integrationsfähigkeit anderer Bussysteme
- breite Unterstützung von Anwendern und Herstellern
- Handhabbarkeit bezüglich Installation und Wartung
- Anpassungsfähigkeit an topologische Gegebenheiten der Anlage
- Robuste Anschlusstechnik
- Zuverlässigkeit (EMV, Redundanz, Fehlerkorrektur)
- Übertragbare Datenrate
- realisierbare Zykluszeiten
- genaue zeitliche Synchronisation verteilter Aktivitäten
- deterministischer Datentransfer zwischen Netzwerkteilnehmern

Die ersten fünf aufgeführten Eigenschaften dienen zur Beurteilung der Leistungsfähigkeit auf der Applikationsebene (Schicht 7) eines Automatisierungs- und Kommunikationssystems und der Flexibilität des Kommunikationssystems (Feldbus) bei der Anpassung an bestehende Automatisierungssysteme oder -geräte. Sie betreffen dabei vor allem den realisierten Funktionsumfang. Die folgenden drei Eigenschaften zielen vornehmlich auf die Beurteilung der generellen Anwendbarkeit eines Kommunikationssystems für eine explizite Automatisierungsanlage und -aufgabe im Feld. Dabei spielen die realisierten Sicherheitsmaßnahmen und die Flexibilität gegenüber örtlich vorhandenen Bedingungen eine übergeordnete Rolle.

Für Automatisierungssysteme im Bereich der Bewegungssteuerung (Motion Control) sind besonders die vier letzten, technischen Eigenschaften wichtig. Dabei bestimmt die übertragbare Datenrate letztlich die realisierbare Komplexität (Datenmenge pro Teilnehmer) und den Ausbau (Anzahl der Teilnehmer) des Gesamtsystems. Die realisierbare Zykluszeit ist neben dem zur Anwendung kommenden Kommunikationssystem auch noch von der Rechenleistung und der internen Datenbus-Beschaffenheit der verarbeitenden Rechnereinheiten abhängig. Sie betrifft vor allem die Zeit, in der der Regelungsalgorithmus komplett abgearbeitet werden kann, also die Zeit für Kommunikation *und* Verarbeitung und damit auch die erreichbare Regelgüte. Neben der realisierbaren Zykluszeit sind eine zeitliche Synchronisation und ein deterministischer Datentransfer von großer Bedeutung.

Abbildung 3.1 veranschaulicht die Auswirkungen von unzureichenden Leistungseigenschaften in diesen Bereichen. Für eine zweidimensionale Bewegungsaufgabe, in

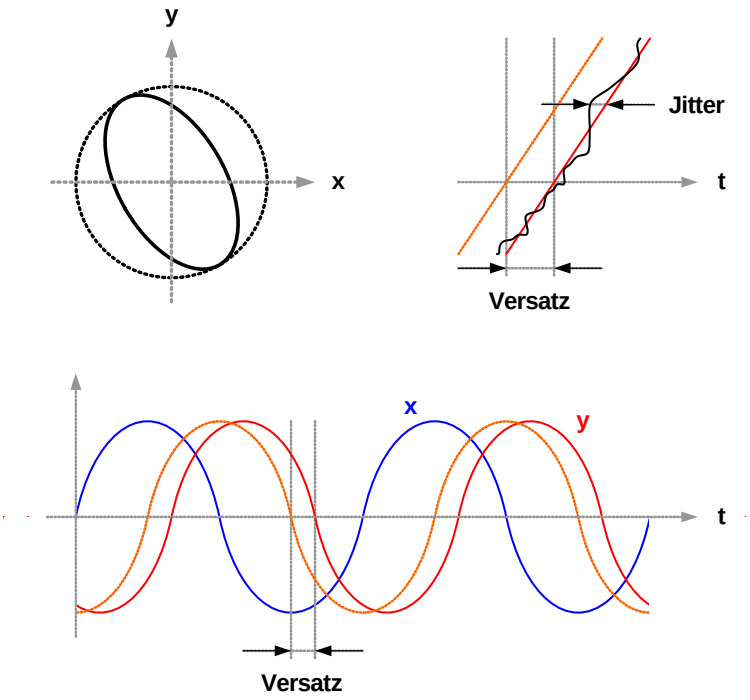



Abbildung 3.1: Auswirkungen von zeitlichem Versatz und Jitter

der eine Kreisbewegung gefahren werden soll, werden sinusförmige Sollwerte für beide Bewegungsachsen auf dem Kommunikationssystem übertragen (unterer Teil, blaue und orange Kurve). Durch die Beschaffenheit des Kommunikationssystems und des Zugriffs sowie durch systematische Unterschiede in der Verarbeitung der Telegramme kommt es dabei effektiv zu einem zeitlichen *Versatz* (rote Kurve). Dieser Versatz führt zu einer Verzerrung der Kreisbahn zu einer Ellipse. Zusätzliche sporadische Schwankungen bei der Übertragung von Datentelegrammen, zeitliche *Jitter*, können auch in der Verfahrbewegung des Roboters ein Zittern erzeugen. Diese beiden Effekte, Versatz und Jitter, sind damit maßgeblich für die erreichbare *Synchronität* bei der Koordination einzelner Bewegungsachsen in einem Steuerungssystem.

## 3.2 Ethernet

Ethernet bezeichnet ein standardisiertes, weit verbreitetes Kommunikationssystem (Definition von Schicht 1 und 2 des ISO/OSI-Schichtenmodells), das über verschiedene Kommunikationsmedien (Koaxialkabel, Lichtwellenleiter, Zweidrahtleitung und Funk) Daten austauschen kann. Als physikalische Basis kommt es vor allem gemeinsam mit der bekannten Kommunikationsprotokollkombination TCP/IP zum Einsatz. Seit einigen Jahren erfährt es eine Erweiterung aus dem Office-Bereich und der Betriebs- und Leitebene heraus hinunter bis in die unterste Feldebene industrieller Anwendungen und schafft damit ein durchgängiges Kommunikationsnetz von der Administrative bis zur Produktion.

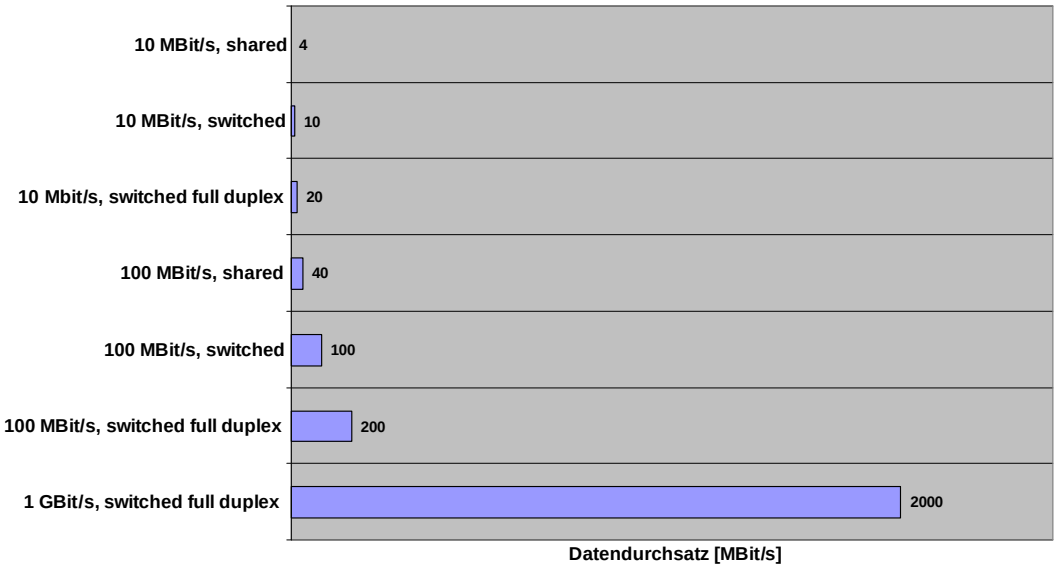



Abbildung 3.2: Durchsatzraten der verschiedenen Ethernet-Technologien [25]

Abbildung 3.2 zeigt die einzelnen Datendurchsätze unterschiedlicher Technologien des Ethernet, die heute allesamt genutzt werden. Für die Industrial-Ethernet-Varianten kommen allerdings bisher nur das Fast- und Gigabit-Ethernet zur Anwendung. Auf diesen basieren alle Industrial-Ethernet-Protokolle und -Anwendungen.

### 3.2.1 Organisationen für Internet/Ethernet

Zur Normung von technischen Spezifikationen und spezifikationsgemäßer Entwicklung von Anwendungen für das Ethernet existieren heute eine Reihe von Organisationen, die es sich zur Aufgabe gemacht haben, das Ethernet in die unterschiedlichen Anwendungsbereiche einzuführen und dort zu etablieren.

Die wesentlichen Organisationen sind hier aufgeführt:

Das **Institute of Electrical and Electronics Engineers**, IEEE [31], ist ein nicht-kommerzielles Normungsgremium für elektrische und elektronische Verfahren. Es kümmert sich auf internationaler Ebene um die Standardisierung verschiedenster Anwendungen aus unterschiedlichen Bereichen wie Raumfahrtsysteme, Computer und Telekommunikation, Biomedizintechnik, Energieversorgung und Unterhaltungselektronik. Dazu ist es in Arbeitsgruppen unterteilt, die in ihrer Gesamtheit mittlerweile mehr als 365.000 Mitglieder haben. Das IEEE hält z.Zt. etwa 900 aktive Standards, während mehr als 400 weitere Standards in Entwicklung sind. Die Kommunikation über das Ethernet ist dabei in ihren unterschiedlichen Ausführungen (Standard, Fast und Gigabit) in den Normen der IEEE 802.3 spezifiziert und dient als Grundlage für die Industrial Ethernet Ansätze, die in diesem Abschnitt beschrieben werden.

Die **Internet Society**, ISOC [32], ist eine internationale nicht-kommerzielle Mitgliedsorganisation, die die Ausbreitung des Internet fördert. Das tut sie durch fi-

nanzielle und juristische Unterstützung der anderen Interessengemeinschaften (s.u.), unter anderem durch Versicherungsschutz für Personen im IETF-Prozess sowie in der Wahrnehmung von Presserepräsentation. Die **Internet Engineering Task Force**, IETF [33], ist eine internationale Interessengemeinschaft, bestehend aus Netzwerkentwicklern, Betreibern, Herstellern und Forschern, die Vorschläge zur Standardisierung im Bereich des Internet umsetzt. 700 Mitglieder organisieren sich dazu in über 80 Arbeitsgruppen, die wiederum zu z.Zt. in acht Arbeitsbereiche (Applications, General, Internet, Operations and Management, Real-time Applications and Infrastructure, Routing, Security, Transport) aufgeteilt sind. Die **Internet Engineering Steering Group**, IESG, besteht aus den Leitern der Arbeitsbereiche der IETF und ist damit zuständig für das technische Management der Arbeiten innerhalb der IETF. Auf diese Weise ist sie maßgeblich an der Entstehung und Weiterentwicklung von technischen Standards beteiligt. Das **Internet Architecture Board**, IAB [34], kümmert sich um die Überwachung der Gesamtarchitektur des Internet. Dabei konzentriert es sich auf die langfristige Planung und Koordination der verschiedenen Bereiche der IETF-Aktivitäten und sorgt damit für architekturbezogene Konsistenz und Integrität. Die **Internet Assigned Numbers Authority**, IANA [35], ist die Registrierstelle für die Aktivitäten der IETF. Viele auf Ethernet basierende (Internet-)Protokolle erfordern die sorgfältige Pflege von Parametern wie TCP-Portnummern und MIME-Typen. Zu diesem Zweck wurde diese Organisation von der IAB ins Leben gerufen, deren Aktivitäten finanziell von der **Internet Corporation for Assigned Names and Numbers**, ICANN [36], unterstützt werden. Die **Internet Research Task Force**, IRTF [37], hat die Aufgabe, die Forschung zur Entwicklung eines zukünftigen Internet voranzutreiben. Dazu beschäftigt sie sich mit Themen aus den Bereichen Internetprotokolle, Anwendungen, Architektur und Technologie. Analog zu der IETF, werden die Arbeitsbereiche innerhalb der IRTF von den Mitgliedern der **Internet Research Steering Group**, IRSG, geleitet, deren Vorsitz von der IAB bestimmt wird.

Das **Computer Emergency Response Team**, CERT [38], erarbeitet Studien zu Sicherheitslücken im Internet und damit verbundenen Gefahren. Außerdem werden Entwicklungstendenzen für netzwerkbetriebene Systeme erforscht und Informationen sowie Trainingsprogramme ausgearbeitet, um die Nutzersicherheit zu erhöhen. Das **Deutsche Network Information Center**, DeNIC eG [39], ist die zentrale Registrierungsstelle für alle Domains unterhalb der Top-Level-Domain „.de“, die im Jahre 1986 in der IANA-Datenbank eingetragen wurde.

Während die oben genannten Organisationen eher den allgemeinen Einsatz von Ethernet und die dazu gehörigen Entwicklungs- und Forschungsaktivitäten hinsichtlich des Internet gestalten, wird die industrielle Anwendung in Form des „industrial Ethernet“ ausschließlich von Firmen vorangetrieben, die in diesem industriellen Sektor Produkte platzieren möchten. Diese Organisationen und Vereinigungen werden im Zuge der Vorstellung der einzelnen Ansätze in den folgenden Abschnitten erwähnt.

### 3.2.2 Industrial Ethernet

Wie zu Beginn des Abschnitt 3 ausgeführt, ist der Einsatz von Kommunikationssystemen in der industriellen Automation, besonders bei Bewegungssteuerungen, an

wesentliche technische Anforderungen geknüpft. Diese bedingen vor allem die *Zuverlässigkeit* und den *Determinismus*, mit der die übertragenen Informationen den verteilten Steuerungsfunktionalitäten zur Verfügung stehen, die erreichbare *Synchronität* koordinierter Aktivitäten in verteilten Geräten und die Möglichkeit des effizienten und schnellen *Austauschs kleiner Datenmengen*. Ethernet ist zunächst nicht geeignet, diese Voraussetzungen für die Anwendung in der industriellen Automation zu erfüllen. Fehlerhaft übertragene Telegramme werden im Empfänger zwar erkannt und dort auch nicht an verarbeitende Schichten weitergeleitet, jedoch erhält der Sender keine Rückinformation über den Empfang des Telegramms, um ggf. eine Wiederholung zu veranlassen; mit steigender Busbelastung nehmen die Übertragungszeiten aufgrund von Kollisionen stark und unvorhersagbar zu, da der Buszugriff über zufällige Ereignisse im Zusammenhang mit dem CSMA/CD realisiert ist. Je nach Netzausbau und Protokollverwendung ergeben sich unterschiedliche Verzögerungszeiten für die Übertragung von Telegrammen, so dass sich koordinierte Abläufe nicht zuverlässig realisieren lassen. Es ist kein Echtzeitverhalten garantiert [27] [21].

Nicht nur die beschriebenen ungünstigen technischen Eigenschaften, sondern auch der Mangel an für die industrielle Automation geeigneten Funktionalitäten der Schicht 7, die in modernen Feldbussen zusätzlich zur Kommunikationsfunktionalität angeboten werden, erschwert den Einsatz von Ethernet in dem Industriesektor. Beispielsweise fehlt ein entsprechendes Objektmodell für die Automatisierungstechnik, in dem sich aus elementaren Objekten wie Merkern, Zählern, Prozessvariablen und Alarmen Funktionsblöcke und Prozessabbilder zusammensetzen lassen. Für die Kommunikation werden in der Automatisierungstechnik neben den Client-Server-Verbindungen häufig auch Publisher-Subscriber-Verbindungen benötigt, um einen zyklischen Datentransfer zu realisieren. Für den Einsatz in der modernen Antriebstechnik ist außerdem ein streng isochroner Datentransfer mit einem Jitter unter einer Mikrosekunde erforderlich.

Abbildung 3.3 zeigt unterschiedliche Klassen vereinfachter Kommunikationsprotokoll-Strukturen, wie sie heute in verschiedenen Anwendungen verwendet werden [48]. *Industrial Ethernet* bezeichnet dabei kein vollständiges, alle Schichten abdeckendes und einheitliches Kommunikationssystem, sondern fungiert als Sammelbegriff für Kommunikations- und Feldbussysteme, die innerhalb der industriellen Automatisierungstechnik auf ein Ethernet (Schichten 1 und 2) nach IEEE 802.3 aufbauen und dieses durch übergeordnete Protokolle, Funktionen oder zusätzliche Hardware erweitern. Standardmäßig werden die Protokolle TCP/IP bzw. UDP/IP für Applikationen verwendet, die keine Echtzeitanforderungen stellen. Es werden Lösungen nach drei Anwendungsklassen unterschieden:

- Lösungen für Klasse 1 verwenden die Protokolle TCP/UDP/IP in ihrer ursprünglichen Form und erweitern diese echtzeitfähig in der darauf aufsetzenden Protokollschicht. Dabei können Zykluszeiten nur im Bereich von $100ms$ erreicht werden.

- Lösungen für Klasse 2 ersetzen die Protokolle TCP/UDP/IP und greifen zur Realisierung von Echtzeiteigenschaften direkt auf die Ethernet-Funktionalität zu. Damit können Protokoll-Laufzeiten reduziert werden, so dass Zykluszeiten

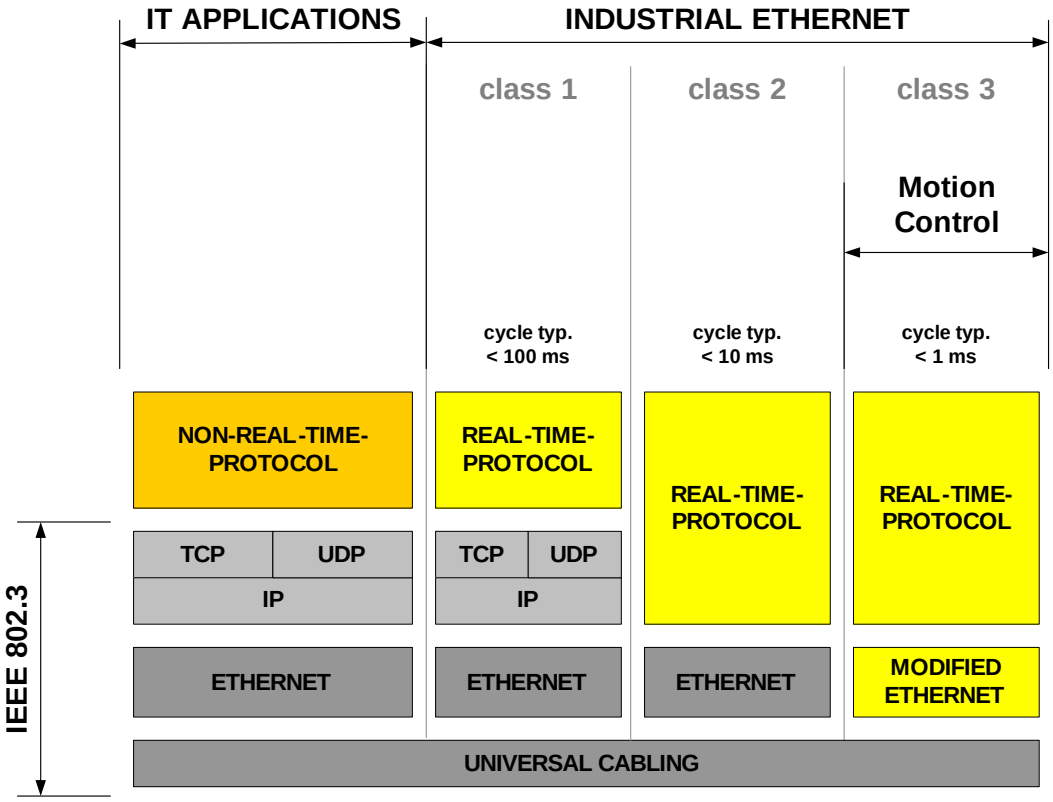



Abbildung 3.3: Unterschiedliche Industrial-Ethernet-Konzepte

im Bereich von $10ms$ möglich werden.

- Lösungen für Klasse 3 modifizieren zusätzlich noch die ursprünglichen Ethernet-Mechanismen, um die Echtzeitfähigkeit weiter zu optimieren. Hiermit können Zykluszeiten unterhalb von $1ms$ realisiert werden.

Da die großen Hersteller und Feldbusorganisationen bestrebt sind, die über die Jahre gewachsenen Gesamtarchitekturen von Feldbussystemen und damit die getätigten Investitionen zu bewahren, wird in der Regel versucht, lediglich die untersten Kommunikationsschichten ihrer eigenen Systeme um Ethernet zu ergänzen. Daraus folgt eine Inkompatibilität der Industrial Ethernet Lösungen zueinander. Die grundsätzlich fehlende Eignung des Ethernet für Echtzeitanwendungen wird dabei durch den Einsatz unterschiedlicher technischer Verfahren aufgearbeitet:

**Netzwerksegmentierung** Durch das Segmentieren des Ethernet-Netzwerkes lassen sich die Kollisionen von Nachrichten einschränken oder sogar komplett vermeiden. Dazu kann jede Komponente über einen separaten *Switch* mit dem Netzwerk verbunden werden (Mikrosegmentierung), so dass bei einem Datentransfer effektiv eine bidirektionale Punkt-zu-Punkt-Verbindung entsteht. Grundlegende technische Details zur Switch-Technologie sind in [21] zusammengefasst. Dadurch, dass jede Komponente einzeln über einen Switch-Port angeschlossen wird oder aber intern einen Switch besitzt, fällt neben den zusätzlichen Hardwarekosten auch ein erheblicher organisatorischer Aufwand zur optimalen Parametrierung der Switche an. Auch die Verzögerungszeiten beim Verarbeiten und Weiterleiten innerhalb der Switche begrenzen dabei die erreichbaren Kommunikationszykluszeiten. Sie sind wesentlich höher als bei der Verwendung von HUBs, die keine Verarbeitung der weiterzuleitenden Telegramme vornehmen müssen. Eine weitere Art der Segmentierung kann über Router erfolgen, die mehrere Teilnehmer (Komponenten) zu einem Segment zusammenfassen. Dieses Verfahren schottet zwei Anwendungsbereiche voneinander ab, so dass eine Kollisionsvermeidung auch ohne weitere einzelne Switche durch die Wahl einer geeigneten Nachrichtentransfer-Koordination (z.B. Zeitschlitze) erreicht werden kann.

**Nachrichtenpriorisierung** Innerhalb der bei der Mikrosegmentierung verwendeten Switche kann gemäß der Ethernet-Erweiterungen nach IEEE 802.1 p und Q eine 3-Bit-Nachrichtenpriorisierung vorgenommen werden. Durch die interne Verwendung mehrerer Telegramm-Zwischenspeicher kann so höherprioren Telegrammen im Netzwerk ein Vorrang bei der Weiterleitung gewährt werden. Allerdings können auf diese Weise nicht die bei der intelligenten Telegrammhandhabung anfallenden Verarbeitungszeiten vermieden werden. Es ergeben sich aufgrund der nicht-präemptiven Bedienung von Telegrammen auch Verzögerungen (in ungünstigen Fällen bis zu $100\mu s$ pro Switch), wenn beispielsweise zeitkritische Telegramme durch die noch andauernde Verarbeitung langer, niederpriorer Nachrichten ausgebremst werden [47].

**Kommunikationsorganisation** Zur optimalen Ausnutzung der Bandbreite des Ethernet und zur Lösung des Problems der Switch-Verzögerungszeiten lässt sich die

gesamte Ethernetkommunikation auch über ein Scheduling auf der MAC-Ebene (Schicht 2) organisieren. Auf diese Weise können Kommunikationsteilnehmer nur nach festgelegten Regeln (zu definierten Zeiten) Telegramme absetzen. Bei einem solchen Verfahren werden damit andere Maßnahmen wie Netzwerksegmentierung und Nachrichtenpriorisierung unnötig, da Kollisionen grundsätzlich vermieden sind. In der Regel erfolgt die Organisation in Form eines Zeitscheibenverfahrens (TDMA), das die Bandbreite des Mediums zyklisch in eine exklusive Echtzeitphase und eine unkritische Phase unterteilt. Die Benachrichtigung der Kommunikationsteilnehmer wird über Token-Passing oder auch Polling realisiert. Diese Funktionalität ist in manchen Fällen auch in einer zusätzlichen Hardware untergebracht.

**Uhrzeitsynchronisation** Um die Aktivitäten innerhalb der Netzwerkteilnehmer in einem Echtzeitkontext zu koordinieren, bietet es sich an, allen Netzwerkteilnehmern eine exakt synchronisierte Zeitbasis zur Verfügung zu stellen. Weit verbreitet ist dazu die *dezentrale* Verwendung des Standards nach IEEE 1588 [45], durch den sich der zeitliche Jitter bei der Ausführung koordinierter Aktivitäten für die Gesamtanwendung minimieren lässt. Zusätzlich ist ein „Kommunikationsplan“ für jeden Teilnehmer erforderlich, der effektiv die mögliche Gesamt-Zykluszeit minimiert. Eine andere Möglichkeit ist die Verwendung einer *zentralen* Zeitorganisation, die über die Synchronisation lokaler Uhren eine Zeitscheibenverwaltung realisiert und damit jede Nachricht auf eine globale Uhr zurückführt.

**Protokollmodifikation** Zur Minimierung der Ausführungszeiten innerhalb der lokalen Telegrammverarbeitung kommt im Austausch oder zur Ergänzung des Standard TCP/IP ein Protokollstack zur Anwendung, der für die Echtzeitanwendung optimierte Funktionalitäten enthält. In der Regel wird zugunsten der Verarbeitungsgeschwindigkeit und der Unabhängigkeit lokal vorhandener Prozessorhardware eine separate Hardware verwendet.

Die Kollisionsproblematik lässt sich bei Ethernet durch die kombinierte Anwendung oben genannter Verfahren beheben. Allerdings können weiterhin zur Abarbeitung der Standardprotokolle TCP/IP und UDP/IP erhebliche Verarbeitungszeiten in den verwendeten Recheneinheiten anfallen. Die reinen Übertragungszeiten der Telegramme bei einer durchschnittlichen Telegrammlänge von 50 Byte Nutzdaten (und ca. 40 Byte Overhead) betragen weniger als $7\mu s$ bei Verwendung von Fast- oder Gigabit-Ethernet. Die Abarbeitungszeiten liegen allerdings, beispielsweise bei Verwendung eines Pentium 166 Prozessors, im Bereich des Fünffachen. Obwohl die Leistungsfähigkeit dieses Prozessors heutzutage nicht mehr aktuell scheint, ist zu bedenken, dass diese Leistungsfähigkeit in jeder Komponente zur Anwendung kommen müsste, was nicht einmal aus Kostengründen realistisch erscheint. Zusätzlich fallen Latenzzeiten an, die sich durch den Weg eines Anforderungstelegramms durch die verschiedenen Schichten der Verarbeitungssoftware ergeben. Selbst bei weiterer Erhöhung der Bitübertragungsrate bestimmt die Verarbeitungszeit innerhalb der Softwareschichten die Grenze für realisierbare Zykluszeiten.

Tabelle 3.1 zeigt eine Zusammenstellung ausgewählter Eigenschaften bekannter und heute verfügbarer Kommunikationslösungen für Industrial (Real-Time) Ethernet. Da-

| | **Buszugriff** CSMA/CD | **TCP/UDP/IP** Std. unterstützt | **Zeitsynch.** IEEE 1588 | **Segmentierung** Kollisionsverm. | **Hardware** Standard | **min.Zykl.zeit** ms | **Jitter** µs |
|---|---|---|---|---|---|---|---|
| **Modbus/TCP** | ja | ja | nein | nicht nötig | ja | < 100 | k. A. |
| **FoundationFieldbus HSE** | ja | nein | nein | nicht nötig | ja | > 100 | k. A. |
| **Ethernet/IP** | ja | ja | nein | über SWITCH | ja | 100 | k. A. |
| **Ethernet/IP (CIPsync)** | ja | ja | ja | über SWITCH | ja | < 10 | < 1 |
| **JetSync** | ja | ja | ähnlich | nicht nötig | ja | k. A. | < 10 |
| **Powerlink** | nein, Zeitfenster | ja, GATEWAY nötig | ähnlich | über ROUTER | ja | > 0,2 | < 1 |
| **PROFInet CBA (NRT)** | ja | verschiedene Kanäle | nein | nicht nötig | ja | 100 | k. A. |
| **PROFInet CBA (RT)** | ja | verschiedene Kanäle | nein | über SWITCH | ja | < 10 | k. A. |
| **PROFInet IO (RT)** | ja, Prio | ja | verwendet | über SWITCH | ja | < 10 | < 10 |
| **PROFInet IO (IRT)** | nein, über ERTEC | ja, GATEWAY nötig | erforderlich | über SWITCH | nein, ASIC | > 0,2 | < 1 |
| **SERCOS III** | nein, Zeitfenster | ja, GATEWAY nötig | eigenes | nicht nötig | nein, FPGA | > 0,03 | < 1 |
| **EtherCAT** | nein, Master-Slave | ja, GATEWAY nötig | verwendet | nicht nötig | nein, ASIC | > 0,03 | < 1 |
| **SynqNet** | nein, Zeitfenster | nein | eigenes | nicht nötig | nein, FPGA | < 0,1 | < 1 |

Tabelle 3.1: Eigenschaften unterschiedlicher Industrial Ethernet Lösungen

bei wurde die Auswahl auf Eigenschaften beschränkt, die sich direkt mit reinen Kommunikationssystemen vergleichen lassen, so dass zusätzliche Funktionalitäten und aufsetzende Protokolle, beispielsweise aus der Steuerungstechnik, nicht aufgeführt sind.

Die Zeitangaben innerhalb der Tabelle sind mit Hilfe der entsprechenden Erklärungstexte zu deuten. Sie sind aus der Interpretation theoretischer Betrachtungen und aus Beispielanwendungen ermittelt worden. In der Regel sind die aufgeführten Zykluszeiten sowie möglicherweise auch die Jitter abhängig vom verwendeten Netzausbau (Switche, HUBs) und der Anzahl verwendeter Teilnehmer und deren zyklisch benötigten Datenmengen. In den verschiedenen Systemen beziehen sich die Synchronisationszeiten auf das Eintreffen von Nachrichten (RxData), das Einhalten exakter Abtastzeitpunkte (Zeitstempel) oder das Weiterleiten durch den Switch (Switch). Die Kommunikationssysteme werden im Folgenden anhand der Tabelleneinträge erläutert.

### 3.2.3 PROFInet

PROFInet ist an sich kein Feldbus auf Ethernet, sondern ein ganzes Automatisierungskonzept, von dem an dieser Stelle nur die reinen Kommunikationsaspekte betrachtet werden. Es existieren zwei Ausprägungen von PROFInet auf der Applikationsebene: PROFInet CBA[1] und PROFInet IO[2] [59]. Die Echtzeitfähigkeit der Kommunikation ist für die Applikation in weiten Teilen skalierbar und teilt sich auf zwei unterschiedliche Datenkanäle auf: Über den Standardkanal können Telegramme ohne Echtzeitanforderungen übertragen werden (Non-Real-Time, NRT). Dabei kommt ein Standard-Ethernet-Netzwerk ohne hard- und softwarebasierte Modifikationen zum Einsatz. In einem Echtzeitkanal werden drei Kommunikationsklassen (Real-Time-Class, RTC) unterschieden, die je nach Netzausbau gemäß der oben aufgeführten Mechanismen genutzt werden können: Übertragung zyklischer Daten ohne spezielle Anforderungen an verwendete Switche (RTC1), Übertragung zyklischer Daten und Alarme unter Verwendung spezieller Switche (RTC2) und Übertragung zyklischer Daten in Motion-Control-Anwendungen unter Verwendung spezieller Switche und einer expliziten Kommunikationsplanung (RTC3 oder Isochronous-Real-Time, IRT). Zur Realisierung des RTC werden generell keine Protokolle auf den Schichten 3 und 4 (TCP/IP bzw. UDP/IP) verwendet, sondern ein modifiziertes Protokoll (RT) auf Schicht 2 (über einen speziellen EtherType angezeigt). Im Falle der RTC3 (IRT) wird die Kommunikationsfunktionalität unter Einbeziehung einer Zeitsynchronisation nach PTP gemäß IEEE 1588 innerhalb eines speziellen Switch-ASICs realisiert, über das jeder Teilnehmer am Netzwerk angeschlossen ist [58]. Auf diese Weise sind Zykluszeiten von $250\mu s$ bei Verwendung von 30 Bewegungsachsen und einer Synchronisation der koordinierten Ausführung von $1\mu s$ erreichbar [60].

[1] PROFInet for Component Based Automation
[2] PROFInet for Input and Output

### 3.2.4 EtherNet/IP

Das Ethernet Industrial Protocol, Ethernet/IP [62], verwendet auch zur Steuerung von Antriebssystemen mit hohen Synchronisationsanforderungen (CIP Motion oder CIP Sync) permanent alle vier Standardschichten des IEEE 802.3 als Basis. Das aufsetzende *CIP* (heute: Common Industrial Protocol, damals: Control and Information Protocol) bildet die Anwendungsschicht. Damit ist eine Möglichkeit geschaffen, neben den Kommunikationssystemen DeviceNet und ControlNet nun auch Ethernet als Übertragungsmedium nutzen und die bestehenden Geräteprofile und Objektbibliotheken weiterverwenden zu können. Ethernet/IP unterscheidet die Kommunikationsarten *explicit* und *implicit*, wobei die erstere das CIP zum Netzwerkmanagement über TCP/IP, letztere für zyklische Daten über UDP/IP überträgt. CIP existiert also neben den bekannten Protokollen FTP, HTTP usw. auf der gleicher Applikationsebene und setzt ein Producer/Consumer Netzwerkmodell um.

Ethernet/IP verwendet eine aktive Sterntopologie, in der jeder Netzwerkteilnehmer über eine Punkt-zu-Punkt-Verbindung mit einem Switch verfügt, welches auf diese Weise Nachrichtenkollisionen verhindert. Die Weiterleitung von Telegrammen geschieht gemäß des VLAN-Tags, so dass Echtzeit-Telegramme bevorzugt werden. Für alle Teilnehmer ist die Verwendung von Standard Ethernet-Chips und Medien möglich. Die Verwendung des Standards IEEE 1588 innerhalb des CIP Sync führt zu einer Verbesserung der Synchronisation verteilter Funktionalitäten innerhalb der Teilnehmer auf unter $1\mu s$. Allerdings sind aufgrund der vielen auch bei der „Echtzeitkommunikation“ durchlaufenen Protokollschichten nur Zykluszeiten $> 10ms$ möglich.

### 3.2.5 Powerlink

Ethernet Powerlink wird von der Ethernet Powerlink Standardization Group (EPSG) betreut [63] und stellt sich aufgrund seiner Funktionen und Bezeichnungen ähnlich wie der IEEE 1394 Standard (FireWire) dar. Die zyklische Protokollabarbeitung teilt sich auf in einen Start-of-Cycle (SoC) und einen isochronen und asynchronen Zeitbereich. Innerhalb des isochronen Zeitbereichs kommt ein reines Master-Slave-Verfahren zum Einsatz. Durch die Organisation der zyklischen Kommunikation in festgelegten Zeitschlitzen werden keine Switche mehr zum Ausschluss von Kollisionen benötigt. Es können die in der Verarbeitungszeit (Durchlaufzeit) wesentlich schnelleren HUBs eingesetzt werden, um das Netzwerk zu erweitern [65]. Ein Abschotten (Segmentieren) der Steuerungsapplikation nach außen geschieht über die Verwendung eines Gateways, das pro Kommunikationszyklus ein einzelnes TCP/IP- oder UDP/IP-Telegramm innerhalb des asynchronen Zeitbereichs übertragen kann. Der Standard IEEE 1588 (PTP) wird verwendet, um für alle Teilnehmer eine gemeinsame Zeitbasis mit einer Synchronisation von $< 1\mu s$ zu realisieren.

Bei Ethernet Powerlink werden zwei Echtzeitarten unterschieden: open mode und protected mode. Im *open mode* für weiche Echtzeit, in dem TCP/IP- und UDP/IP-Telegramme ungehindert parallel zu den Echtzeit-Telegrammen übertragen werden, können dabei Zykluszeiten im Millisekundenbereich und Jitter unterhalb von $10\mu s$ erreicht werden. Im *protected mode* für harte Echtzeit werden Zykluszeiten unterhalb

von $100\mu s$ bei einem Jitter von weniger als $1\mu s$ erreicht. Beispielsweise können in einem Zyklus von $1ms$ 44 Teilnehmer mit je 46 Byte Ist- und Sollwertdaten bedient werden [64]. Dazu trennt dann ein Router den Steuerungsbereich von dem IT-Bereich und integriert von außen zu kommunizierende TCP/IP-Telegramme innerhalb eines Zeitschlitzes im Echtzeitzyklus.

### 3.2.6 JetSync

JetSync ist eine auf Ethernet und TCP/IP basierende Kommunikationsschnittstelle und kommt im Komplettsystem JetWeb [67] der Firma Jetter bei der Antriebssteuerung zum Einsatz. Das Funktionsprinzip gründet sich auf der Synchronisation der in den verteilten Geräten vorhandenen Uhren. Das Verfahren zur Synchronisation ist dabei ähnlich dem Standard IEEE 1588, jedoch mit geringerer Genauigkeit, so dass eine Koordination von Aktivitäten innerhalb der verteilten Geräte mit einer Schwankung von $10\mu s$ möglich ist. Zur Nutzung dieses Synchronisationsmechanismus erhalten die Telegramme bei ihrer Übertragung Zeitstempel, die ihre Koordination innerhalb der Regelung oder den Zeitpunkt der Messung genau spezifizieren. Auf diese Weise ist die Ausführung von verteilten Aktivitäten unabhängig vom aktuellen Datenaufkommen sowie dem Empfangszeitpunkt zugehöriger Datentelegramme, sofern sie noch rechtzeitig empfangen wurden. Ein Zugriff über das Internet (z.B. als Download) während des Synchronlaufs ist vollständig möglich.

### 3.2.7 Modbus/TCP (IDA)

Modbus/TCP ist die Umsetzung des bekannten Modbus-Anwendungsprotokolls (seit 1979 verwendet) auf ein TCP/IP-Netzwerk [53]. Hierbei kommt die gesamte TCP/IP-Protokollfamilie mitsamt Standard-Ethernet ohne Veränderung innerhalb unterer Schichten zur Anwendung. Modbus/TCP ist wie MODBUS ein reines Request/Reply-Protokoll und nutzt im TCP-Frame den Port Nr. 502. Zur Realisierung eines echtzeitfähigen Publish-Subscribe-Mechanismus wurde auch das RTPS [54] als zusätzliche Echtzeiterweiterung definiert. Das RPTS setzt auf UDP/IP auf und bietet die beiden Kommunikationsmodelle Publish-Subscribe und Composite-State-Transfer (CST) an. Allerdings ist RTPS im Gegensatz zu Modbus/TCP nicht verbreitet.

### 3.2.8 SERCOS III

Das Serial Real-Time Communications System, SERCOS [66], ist in seiner dritten Generation nun auf dem physikalischen Ethernet-Medium anwendbar. Die Telegramme können per Lichtwellenleiter oder Fast-Ethernet übertragen werden, das mit Standard Ethernet-Hardware und einem zusätzlichen, programmierbaren Logikbaustein realisiert ist. Dabei bleiben alle ursprünglichen Kommunikationseigenschaften von SERCOS II erhalten bzw. werden noch zugunsten neuester Entwicklungen im Bereich der Antriebstechnik verbessert.

Das bewährte zeitschlitzbasierte Kommunikationsprotokoll wird verwendet, welches durch die zyklische Übermittlung von Master-Cycle-Telegrammen (MST), Amplifier-Telegrammen (AT) und Master-Data-Telegrammen (MDT) die Echtzeitkommunikation abwickelt. Neu an SERCOS III ist der zusätzliche IP-Datenkanal, in dem standardmäßige TCP/IP- oder UDP/IP-Telegramme „parallel“ zur zyklischen Echtzeitkommunikation übertragen werden können. Auch ein Datenquerverkehr ist nun möglich. Die Zeit- und Ereignissynchronisierung wird bei SERCOS III nicht über den Einsatz des Standards IEEE 1588 vorgenommen, sondern ist, wie bisher, über den Empfang der ohnehin versendeten MDTs realisiert. SERCOS III erzielt damit eine minimale Zykluszeit von $31,25\mu s$ bei 8 verwendeten Antrieben mit jeweils 8 Byte Ist- und Sollwertgröße. Es besteht durch Verwendung einer Doppelringstruktur als Topologie die Möglichkeit, ein redundantes Verhalten zu erzielen, das auch bei Unterbrechung einer Verbindung die Funktion des Netzwerks und der Steuerung aufrechterhält. Da keine HUBs und Switche verwendet werden, konnte der Jitter auf unter $100ns$ minimiert werden.

### 3.2.9 SynqNet

SynqNet ist eine digitale Antriebsregelungs-Schnittstelle, die seit dem Jahre 2002 in Produkten für die Automatisierungstechnik eingesetzt wird. Der Hersteller von SynqNet, Motion Engineering, sieht sein Produkt als erstes verfügbares 100BaseT-Netzwerk, das alle Leistungsvorteile für die Anwendung eines zentralisierten Regelungsmodells aufweist. SynqNet nutzt die Kabel und die physikalische Schicht des IEEE 802.3 Standards, modifiziert allerdings die MAC-Ebene der Schicht 2. Hiermit werden besonders auf erreichbare Zykluszeit getrimmte Leistungseigenschaften erreicht, da diese in dynamischen Antriebssystemen über Steifheit und Positionsgenauigkeit entscheiden können. Die Nutzung der Voll-Duplex-Leitungspaare ermöglicht eine effektive Datenrate von $2 * 100MBit/s$. Ähnlich wie bei SERCOS III ist auch der Aufbau einer redundanten, robusten Kommunikationsstruktur möglich. Dazu werden pro Teilnehmer jeweils zwei Ethernet-Verbindungen verwendet. Es sind keine TCP/IP oder andere Ethernet-Nachrichten verwendbar, so dass für die Standard-Ethernet-Kommunikation ein zusätzlicher Kommunikationsbus verwendet werden müsste.

Während für ein Standard-Ethernet-Datagramm mindestens 74 $(28 + 46)$ Bytes verwendet werden, reduziert sich dieses bei SynqNet auf 24 Bytes. Der ursprüngliche CSMA/CD-Mechanismus wird durch ein synchrones, zeitbasiertes Zugriffsverfahren ersetzt. SynqNet verwendet PLL-Techniken, um die unabhängigen Zeitgeber jedes Netzwerkteilnehmers zu synchronisieren. Damit werden die auftretenden Jitter auf unter $1\mu s$ begrenzt. Außerdem sorgen spezielle Algorithmen dafür, dass auch ein systematischer Kommunikationsversatz zwischen den Teilnehmern auf unter $20ns$ minimiert wird. Diese Versatz- und Jitter-Eigenschaften werden für eine beliebige Anzahl von Teilnehmern und für ein beliebiges Datenaufkommen garantiert. Beispielsweise kann für geschlossene Regelkreise mit vier unabhängigen Achsen eine Zykluszeit von $25\mu s$ erreicht werden; 32 Achsen werden in einem Zyklus von $100\mu s$ bedient.

### 3.2.10 EtherCAT

Das System Ethernet for Control Automation Technology, EtherCAT [56], geht mit seiner Funktionsweise einen anderen Weg als die übrigen Industrial-Ethernet-Ansätze. Die Topologie ist nicht stern-, sondern ringförmig (offen), so dass Telegramme generell von einem Teilnehmer zum nächsten weitergeleitet werden. Anstatt für jeden Busteilnehmer einzelne Telegramme für die Prozessdatenkommunikation (Sollwerte und Istwerte) zu verwenden, versendet der EtherCAT-Master in der Regel nur *ein* Telegramm pro Kommunikationszyklus. In den Slaves kommt ein ASIC zur Anwendung, das ein $64kByte$ DPRAM und DMA-Transfers bedient und damit die gesamte Organisation und Kommunikation steuert, so dass es generell keine Abhängigkeit von CPU-Leistungsfähigkeit und Protokollabarbeitung gibt. Im Durchlauf durch jeden Teilnehmer (Slave) liest dieser die für ihn bestimmten Soll- und Steuerdaten heraus und schreibt seinerseits Ist- und Statusdaten an entsprechenden Bitpositionen hinein. Dabei wird das Telegramm nur um wenige Nanosekunden verzögert ($60ns$ pro Gerät [50]). Der Master empfängt das zuvor von ihm versendete Telegramm auf dem Rückkanal der Vollduplex-Ethernetleitung und erhält damit in jedem Zyklustakt ein vollständiges Abbild der Prozessdaten.

Auf diese Weise wird unter anderem der unangenehm große Protokoll-Overhead, der vor allem bei kleinen Nutzdatenmengen beim Versenden von Ethernet Datagrammen vorherrscht, vermieden; Kollisionen werden prinzipiell ausgeschlossen, so dass keine spezielle Switch-Technologie angewendet werden muss. Mit dem beschriebenen Verfahren und der Anwendung des IEEE 1588 Standards (PTP) lassen sich die Aktivitäten der EtherCAT-Teilnehmer auf weniger als $100ns$ synchronisieren [57]. Es sind auch bei Einsatz vieler Teilnehmer Kommunikationszykluszeiten von weniger als $100\mu s$ möglich. Trotzdem ist zu jeder Zeit nach wie vor eine Standard-TCP/IP-Kommunikation auch über Standard-Hubs und Switche möglich.

## 3.3 Weitere Kommunikationssysteme

Die Industrial-Ethernet-Ansätze dominieren den Markt und die Medien bezüglich neuer Anwendungen für den industriellen Kommunikationssektor. Es existieren trotzdem weitere leistungsfähige Kommunikationssysteme, die sich auch ohne Ethernet-Bezug in Anwendungen der industriellen Automation behauptet haben sowie weiterhin erkämpfen.

### 3.3.1 IEEE1394 Standard (FireWire)

1986 wurde „FireWire“ von Entwicklern der Fa. Apple Computer als Kommunikationssystem für multimediale Anwendungen konzipiert und 1995 als Standard IEEE 1394 verabschiedet. Seither wird der ursprüngliche Standard zwecks Anpassung an technologische Weiterentwicklungen und der Erschließung neuer Anwendungsfelder modifiziert und verbessert.

### Normungssituation

Analog zur Spezifikation von Ethernet beschreibt die FireWire-Spezifikation „nur“ mögliche Übertragungsraten, physikalische Signale, angebotene Dienste, spezifikationskonforme Telegramme und zulässige Parameter; nicht aber, in welcher Weise zu ihrer Verwendung auf Funktionalitäten zugegriffen werden soll. Damit definiert der FireWire-Standard eine Schnittstelle auf der Netzwerk-Treiberebene. FireWire ist deshalb von unterschiedlichen Herstellern erhältlich, jeweils aber mit zueinander inkompatibler Treiberschnittstellenimplementierung (im Gegensatz zu SynqNet, welches unabhängig vom Hersteller mit einer einheitlichen API erhältlich ist). Um den leistungsfähigen Standard für reale Anwendungen nutzen zu können, werden Protokollimplementierungen benötigt, die für jeweilige Aufgabenfelder spezifiziert sind und auf einer einheitlichen Treiberebene aufsetzen. Bis heute sind FireWire-Anwendungen nur als proprietäre Lösungen erhältlich (z.B. Fa. Nyquist), was die Ausbreitung einschränkt und auch Interoperabilitätsprobleme zu anderen Herstellern nicht löst.

### Anwendung in der Industrie

Die Einführung von FireWire in die Industrieautomation wird von der europäischen 1394 Automation Gruppe, zusammen mit der internationalen 1394 Trade Association (mehr als 170 Mitglieder, unter anderen Sony, Texas Instrument, Microsoft, Philips) vorangetrieben. Entwickelt wurde bereits ein offener Standard (1394AP) für den Datenaustausch mittels FireWire in der Industrieautomation [69] [71]. Der Standard beruht auf den in [22] definierten Mechanismen für den isochronen Datentransfer. Zurzeit wird durch die Umsetzung des Standards in Produkte dessen Verbreitung in der Automation vorangetrieben.

### Eignung für Motion-Control-Anwendungen

Eine essenzielle Anforderung an die Datenübertragung in der Industrieautomation und insbesondere im Bereich „Motion Control“ ist die strikte Synchronität der Steuerdaten. Voraussetzung auf Busebene dafür ist eine so genannte isochrone Übertragung, das heißt, dass Daten mit einer konstanten Frequenz versendet werden. Diese Eigenschaft lässt sich in Kommunikationstechnologien ohne isochrone Übertragung (z.B. Ethernet), wenn überhaupt, nur mit Aufwand nachrüsten. FireWire bietet von Haus aus einen isochronen Übertragungsmodus mit einer Taktrate von $8kHz$ und ist deshalb für die genannten Anwendungen besonders gut geeignet. Außerdem ist es mit FireWire möglich, zusätzlich zu Echtzeitdaten auch Informationen mit höherem Datenratenbedarf, wie z.B. Bilddaten zur optischen Kontrolle eines Herstellungsprozesses, über eine gemeinsame Leitung zu übertragen.

### Verfügbarkeit von Komponenten

FireWire nutzt Komponenten, die in Massenmärkten wie der Computerindustrie eingesetzt werden. Der 1394AP-Standard ist eine reine Softwareimplementierung und erfordert keine spezielle Hardware. All dies bewirkt, dass sich FireWire mit geringen Kosten in Geräte der Industrieautomation integrieren lässt, die unterhalb derer vergleichbarer Technologien liegen. Die Offenheit des Standards spielt hierbei auch eine entscheidende Rolle [70].

Zahlreiche Kommunikationsmodule sind auf Basis von PCI- und PCMCIA-Karten verschiedener Hersteller erhältlich. Allerdings sind passende Treiber dafür vor allem für Windows-Betriebssysteme realisiert und decken damit lediglich die Anwendung im Multimediabereich ab. Einen direkten und deterministischen Zugriff auf schnelle Funktionen der FireWire-Hardware bieten nur wenige Hersteller, da die Realisierung dazu auf Echtzeitbetriebssystemen (QNX, VxWorks, RTLinux) erfolgen muss (z.B. Fa. Mindready [72]). Auch Mikrocontroller- und DSP-Systeme ohne Betriebssysteme werden mit einer Reihe von Lösungen bedient (z.B. Fa. Orsys [73]). Die Untersuchung möglicher Plattformen zur Realisierung einer Kommunikations-Infrastruktur, wie sie im SFB 562 Verwendung finden sollte, ergab Beschränkungen aller angebotenen Lösungen, die zum Teil aus der Verwendung aktuell verfügbarer Komponenten herrührten. Heute sind Komponenten erhältlich (z.B. der LLC CeLynx von TI), die neben ausreichenden deterministischen Leistungseigenschaften auch über eine standardisierte Schnittstelle (OHCI[3]) zur Programmierung verfügen.

**Firmen und Produkte**

Mitgliedsfirmen der 1394 Automation Gruppe bieten bereits vielfältige Produkte an. Exemplarisch seien Motion Controller von Nyquist [85], Antriebe von Stöber und SSD Drives [84] sowie I/O-Module von Nyquist und Wago genannt. Vorrangige Anwendungen sind alle Systeme, bei denen es auf eine hochpräzise Steuerung ankommt. Als ein Beispiel seien Waferhandlingsysteme in der Halbleiterindustrie genannt [70].

Im Bereich multimedialer Anwendungen bei besonders hohen Bandbreitenanforderungen im Automobil wird der Standard IDB[4]-1394 verwendet. Er definiert die Kabel, Verbinder und die höheren Protokollschichten, die notwendig sind, IDB-1394-Geräte miteinander zu verbinden. Dabei ergänzt er die existierenden Standards IEEE 1394-1995, IEEE 1394a-2000 und IEEE p1394b zum Betrieb innerhalb des Fahrzeug-Innenraums und zum Anschluss tragbarer Multimediageräte. Ein Beispiel für eine erfolgreiche Anwendung dieses Protokolls ist für Renault in [74] zu finden.

### IEEE 1394-1995

In den folgenden Absätzen finden sich technische Erläuterungen zum Standard, die in ähnlicher Form bereits größtenteils auch in [21] zusammengetragen wurden. Die Quelle ist in jedem Fall die Spezifikationsbeschreibung des Standards [68] sowie der Standard selbst [75] [76] [77] [78]. Da die in Abschnitt 4 dieser Arbeit beschriebenen softwaretechnischen Entwicklungen funktional stark auf den FireWire-Standard bezogen sind (besonders die Realisierung des IAP, s. Abschnitt 4.3), erfolgt hier eine umfangreiche Darstellung.

**Topologie**

FireWire verbindet maximal 63 Knoten (Teilnehmer) pro Segment und insgesamt 1023 adressierbare Segmente miteinander. Die Netzstruktur besteht dabei physika-

[3] OHCI = Open Host Controller Interface
[4] IDB = Intelligent transportation systems Data Bus

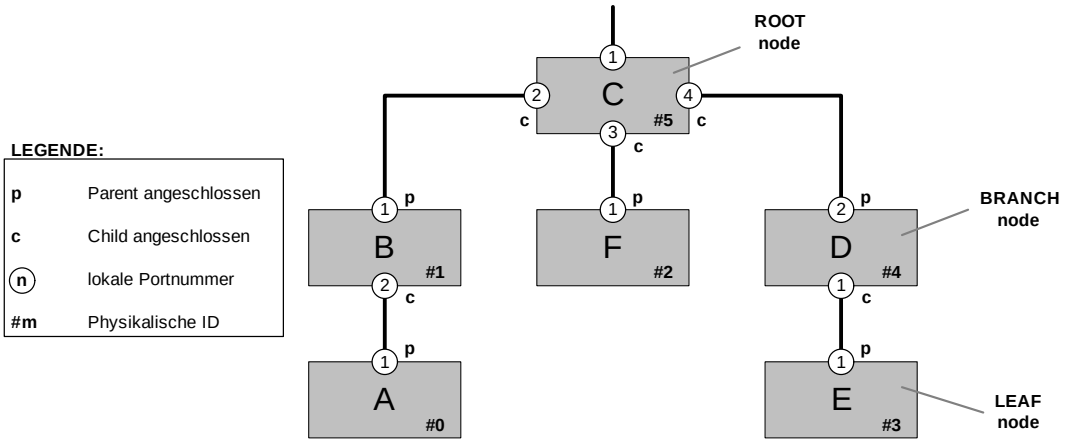



Abbildung 3.4: Topologie eines FireWire-Netzes

lisch aus Punkt-zu-Punkt-Verbindungen der Knoten, aus denen logische Bäume mit einer maximalen Tiefe von 15 Zwischenstationen gebildet werden. Abbildung 3.4 zeigt ein Beispiel für eine Topologie-Konfiguration. Es werden drei Typen von Knoten unterschieden: Wurzel (root), Zweig (branch) und Blatt (leaf). Ein im Netz übertragenes Telegramm wird über einen Port (in der Abbildung für jeden Knoten nummeriert) empfangen und automatisch an alle übrigen Ports des Knotens weitergeleitet. Auf diese Weise gelangt jedes Telegramm zu jedem Knoten im Netz. Der Wurzelknoten besitzt eine Sonderrolle. Er ist in der Regel für die Verwaltung (bus management) und das Timing (cycle management) verantwortlich.

Wenn Knoten ein- oder ausgeschaltet werden oder die Verbindung eines Knotens zum Netz unterbrochen oder neu hinzugefügt wird, erfolgt ein automatischer Bus-Reset. Dieser bewirkt, dass sich das Netzwerk und seine Topologie neu identifiziert. Die Identifizierung geht dabei in drei zeitlich festgelegten Phasen vor sich:

**Bus Initialisierung** Ein Bus-Reset dauert $167\mu s$, damit sichergestellt ist, dass ein Knoten, der beim Versenden unterbrochen wurde (dies aber nicht registriert) vom Bus-Reset sicher erfasst wird. In der anschließenden Phase „Bus Initialisierung“ werden sämtliche Topologieinformationen in allen Knoten gelöscht, so dass ein „flaches“ Netz entsteht. Anschließend überprüfen die Knoten das Vorhandensein anderer Knoten an ihren Ports. Ist nur ein Port mit einer Verbindung zu einem anderen Knoten belegt (aktiv), so ist der Knoten ein Leaf-Knoten; sonst ein Branch-Knoten.

**Tree Identification** In dieser Phase erfolgt die Bildung der Baumstruktur und die Festlegung des Root-Knotens. Dazu signalisieren zunächst alle Leaf-Knoten ein `PARENT_NOTIFY` an ihrem aktiven Port. Die angeschlossenen (Branch-)Knoten signalisieren immer ein `CHILD_NOTIFY`, wenn maximal an einem ihrer aktiven Ports *kein* `PARENT_NOTIFY` erkannt wurde. Dieser Port wird dann im (Branch-)Knoten lokal als „child“ (c) markiert. Sobald ein Leaf-Knoten ein beantwortendes `CHILD_NOTIFY` an seinem Port detektiert, markiert er seinen Port als „parent“ (p), da dort ein übergeordneter Knoten angeschlossen ist. Diese Prozedur läuft so lange, bis ein einzelner Knoten nur `PARENT_NOTIFY` an seinen aktiven

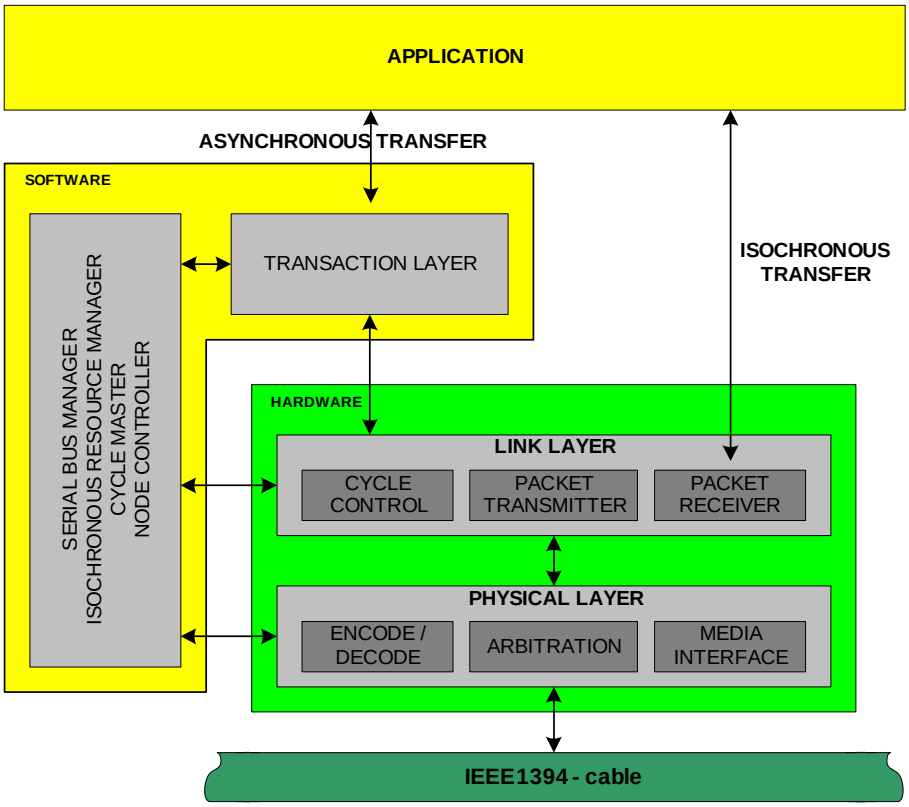



Abbildung 3.5: Schichtenmodell des FireWire-Standards

Ports liest. Er ist damit automatisch der Root-Knoten. Wenn unter bestimmten Umständen zwei Knoten sich gegenseitig ein `PARENT_NOTIFY` signalisieren, so ziehen sie ihre Signale für eine zufällige Zeitspanne zurück und signalisieren erneut, wenn daraufhin die Signalleitung unbenutzt ist[5]. Diese Phase dauert in der Regel weniger als $10\mu s$.

**Self Identification** In dieser Phase wird jedem Knoten eine eindeutige Knoten-ID (physical ID) zugeordnet. Dazu leitet der Root-Knoten den Transfer von Self-ID-Telegrammen ein. Dabei wird ein `SELF_ID_GRANT` jeweils an dem aktiven Port mit der kleinsten Nummer signalisiert und weitergeleitet, bis ein erster Leaf-Knoten erreicht ist. Dieser erhält die Physical-ID 0 und sendet seine maximale Übertragungsgeschwindigkeit, seine Leistungsspezifikation und weitere Informationen innerhalb eines Telegramms. Der nächste Knoten verfährt ebenso, bis der Root-Knoten als letzter die Self-Identification abschließt. Pro Knoten dauert diese Phase ca. $1\mu s$. Anschließend werden die ermittelten Topologieinformationen an alle Knoten übertragen.

### Schichtenmodell

Die einzelnen Funktionalitäten des Standards sind in verschiedenen Soft- und Hardwareschichten realisiert, die von allen Knoten verwendet werden und in Abbildung 3.5

[5] Es ist möglich, verzögert an der Tree Identification teil zu nehmen. Damit kann ein einzelner Knoten seine Identität als Root-Knoten erzwingen.

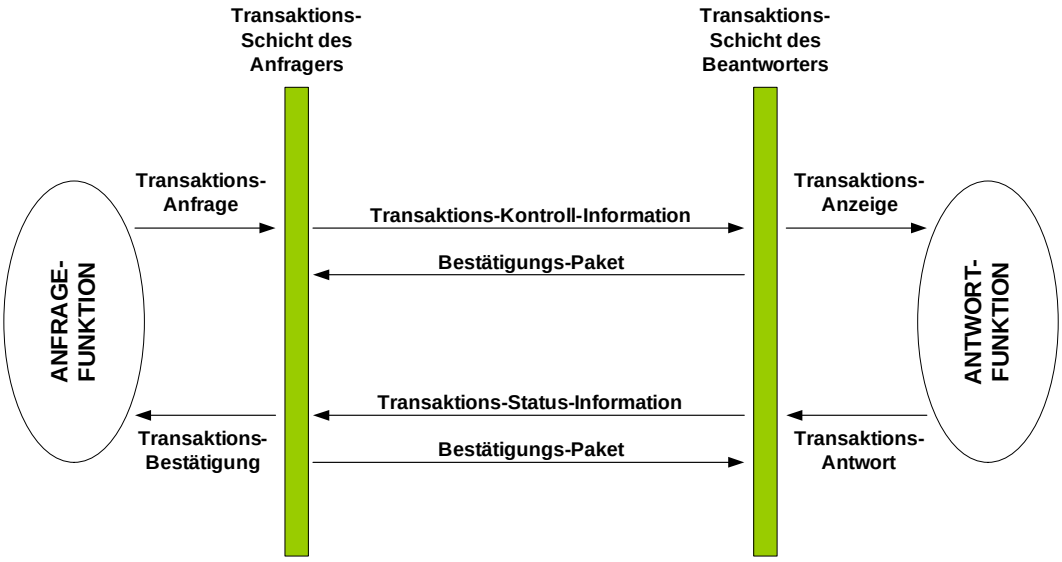



Abbildung 3.6: Aktivitäten des Transaction-Layers

im Schichtenmodell eingeordnet sind:

**Kabel** Das FireWire-Kommunikationsmedium besteht aus 3 Leitungspaaren: für Strobesignale, Datensignale und optional Versorgungsspannung. Aus den Daten- und Strobesignalen werden auch die Taktsignale generiert. Die Leitungslängen sind dabei zunächst auf $4,5$ m begrenzt.

**Physical Layer Controller (PHY)** Der PHY ist in der Regel als separater integrierter Baustein verfügbar. Er übersetzt elektrische Signale auf den Leitungen in von Recheneinheiten verarbeitbare Datenpakete und umgekehrt. Dabei übernimmt er auch die Bus-Initialisierung und die Organisation des Buszugriffs im laufenden Betrieb. Als Repeater gibt er über einen Port einkommende Telegramme an allen aktiven Ports wieder aus.

**Link Layer Controller (LLC)** Der LLC wird in der Regel ebenfalls als integrierter Baustein verwendet. Über seine Schnittstelle zum PHY tauscht er Datenpakete aus und stellt Fehlerkorrekturmaßnahmen und automatische Bestätigungsaktivitäten zur Verfügung. Der LLC leitet erkannte Busaktivitäten über Interruptleitungen an verarbeitende Software-Ebenen weiter.

**Transaction Layer** Diese in Software-Ebene realisierte Funktionsschicht ist allein für die Behandlung asynchroner Telegramme (s.u.) zuständig (Abbildung 3.6). In ihr wird die Auswertung von Fehlerereignissen und die Organisation geschlossener Transaktionen durchgeführt. Außerdem übernimmt sie auch das isochrone Ressource-Management. Letztlich bleibt es allerdings dem Anwender überlassen, ob die Funktionalitäten dieser Schicht realisiert werden.

**Serial Bus Manager und Isochronous Resource Manager** Auch diese Schicht ist komplett in Software realisiert. Der Isochrone Ressourcemanager (IRM) übernimmt die Zuteilung isochroner Bandbreite, während der Serielle Busmanager

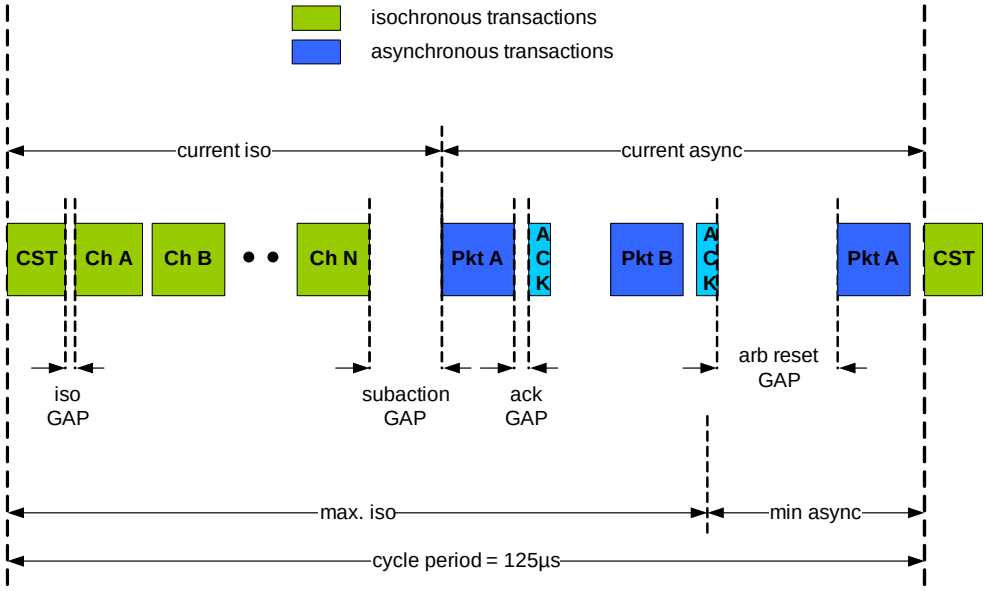



Abbildung 3.7: FireWire-Übertragungszyklus

(SBM) Topologieinformationen und Geschwindigkeitskarten des aktuellen Netzwerks verwaltet. Er optimiert auch die zur Synchronisation verwendeten GAP-Zeiten gemäß des aktuellen Systemausbaus und regelt das Power-Management.

### Kommunikationszyklus

Abbildung 3.7 zeigt den FireWire-Übertragungszyklus. FireWire stellt unabhängig von der gewählten Datenrate (100, 200 oder $400 MBit/s$) stets eine konstante Zykluszeit von $125 \mu s$ zur Verfügung. Während dieses Zyklus erfolgt nacheinander zunächst der isochrone und anschließend der asynchrone Transfer für alle am Bus angeschlossenen Knoten. In der Abbildung sind die wichtigsten GAP[6]-Definitionen dargestellt. Nach ihnen richtet sich die Erkennung der aktuellen Transfers und der Arbitrierungsberechtigung der Knoten am Bus.

Der **isochrone** Datentransfer dient zur Übertragung von Echtzeitdaten. Es ist ein unbestätigter Dienst, so dass bei Übertragungsfehlern keine Wiederholung der Daten erfolgt. Aus 64 möglichen isochronen Kanälen kann jeder Knoten diejenigen festlegen, die er am Bus mithören möchte. Das Senden auf einem Kanal ist jeweils nur einem Knoten gestattet, der dies beim IRM[7] (i.d.R. der ROOT-Knoten) zuvor beantragt. Dieser reserviert daraufhin die nötige Bandbreite für den isochronen Transfer, die dem entsprechenden Teilnehmer in jedem Zyklus garantiert zur Verfügung steht. Eingeleitet wird der isochrone Transfer in jedem Zyklus über das Versenden eines CST[8],

[6] die GAP-Zeiten entsprechen unterschiedlichen Verzögerungszeiten
[7] IRM = Isochronous Resource Manager
[8] CST = Cycle Start Telegram

für das der CM[9] (im ROOT-Knoten) verantwortlich ist. Ab dem CST stehen dann für den isochronen Transfer maximal $100\mu s$ (iso GAP) zur Verfügung; während dieser Zeit werden die isochronen Telegramme mit einem Abstand von $0,05\mu s$ (iso GAP) auf den Bus geschrieben. Für den asynchronen Transfer verbleiben damit noch mindestens 20% des Zyklus. Eine mögliche Verschiebung des Zyklusendes wirkt sich bei FireWire als Jitter der isochronen Telegramme aus. Dies geschieht, wenn ein asynchrones Telegramm zwar innerhalb eines Zyklus, allerdings so weit am Zyklusende versandt wird, dass das Telegrammende den nächsten regulären Zyklusstart nach hinten verschiebt. Der Verzug wird aber im CST mit übertragen, so dass ein Rückrechnen zur genauen Bestimmung der zeitlichen Zusammenhänge erfolgen kann.

Der **asynchrone** Transfer ist in der Regel als eine gerichtete Übertragung mit Fehlerprüfung ausgeführt. Dabei ist die Übertragung vierteilig: Zunächst erfolgt die Anfrage, diese wird von dem Zielknoten zunächst innerhalb von $0,05\mu s$ bestätigt (ACK, ack GAP). Sobald die Anfrage im Zielknoten bearbeitet ist, erfolgt eine Antwort, die vom Anfrager wieder mit einer Bestätigung quittiert wird. Der asynchrone Transfer kann aber auch als Broadcast-Übertragung ausgeführt sein; dann entfällt allerdings die Bestätigung. Als Besonderheiten des asynchronen Transfers gelten „Split-Transactions“, bei denen zwischen einer Bestätigung und einer Antwort andere Knoten bereits Transaktionen initiieren können. Sollte aber ein Knoten schnell sein, so kann er die Antwort auch direkt an seine eigene Bestätigung anhängen („Concatenated Transaction“) und so die Transaktion ohne erneute Arbitrierung beschleunigen. Um sicher zu stellen, dass nicht ein Knoten den asynchronen Transfer dominiert, existiert bei FireWire ein „Fairness“-Mechanismus. Zwischen zwei aufeinander folgenden asynchronen Telegrammen muss ein Zeitabstand von mindestens $10\mu s$ liegen (subaction GAP). Jeder Knoten darf aber erst dann erneut ein asynchrones Telegramm verschicken (unabhängig von Zyklusvorgängen), wenn ein Zeitintervall von $20\mu s$ verstrichen ist und damit klar ist, dass kein anderer Knoten den Erst-Versand im aktuellen Fairness-Intervall durchführen möchte.

Die Sendeberechtigung für den Bus erhalten die Knoten nach erfolgreicher **Arbitrierung**. Der Arbitrierungsvorgang berücksichtigt das Garantieren isochroner Bandbreite sowie die Einhaltung des Fairness-Intervalls beim asynchronen Transfer. Je nach momentanem Buszustand (isochroner Transfer bzw. asynchroner Transfer) und individuellem Teilnehmerzustand (isochrone bzw. asynchrone Telegramme versendet oder nicht), gelten unterschiedliche GAP-Zeiten, die von allen Teilnehmern abgewartet werden müssen, bevor eine Arbitrierung eingeleitet wird. Auf diese Weise wird sichergestellt, dass zunächst der isochrone Transfer (iso GAP), dann der asynchrone Transfer (subaction GAP) und dann erst ein weiterer asynchroner Transfer des selben Knotens (arb reset GAP) zugelassen wird. Je nach örtlicher Nähe zum ROOT-Knoten erreicht die Busfreigabe einen Knoten unterschiedlich schnell. Dieser kann zuerst mit einer Datenübertragung beginnen. Die GAP-Zeiten können auch weiter optimiert werden, wenn z.B. klar ist, dass die Anwendungsausdehnung nur kurze Signallaufzeiten zwischen den Knoten verursacht und die doppelte Signalisierungszeit zum elektrisch am weitesten entfernten Knoten über die Wartezeiten abgedeckt ist.

[9] CM = Cycle Master

### IEEE 1394a

Neben einigen Fehlerkorrekturen, die z.B. das elektrische Entprellen beim Verbinden neuer Knoten an den Bus betreffen, sind im a-Standard wesentliche Verbesserungen für den Kommunikationszyklus spezifiziert worden. Besonders Latenzzeiten während des Arbitrierungsvorgangs, und damit der Zugriff der Knoten auf den Bus, wurden dadurch optimiert.

Auf ein empfangenes asynchrones Telegramm folgt im Abstand von $0,05\mu s$ ein Acknowledge-Telegramm vom Empfänger. Nach bisheriger Spezifikation musste jeder Knoten vor dem Senden einer neuen Anfrage oder Antwort einen Subaction-GAP von $10\mu s$ abwarten. Der Mechanismus „**Ack-accelerated Arbitration**“ bewirkt, dass dieser GAP nun nur noch einmalig zwischen isochronem und asynchronem Transfer eingehalten werden muss. Mit dieser Neuerung kann jeder Knoten sofort nach einem beliebigen ACK-Telegramm arbitrieren. Dadurch erhöht sich die Anzahl bedienbarer asynchroner Knoten pro Zyklus.

Zusätzlich zu den „Concatenated Transactions“ des ursprünglichen Standards wurde der „**Fly-by Concatenation**“-Mechanismus spezifiziert. Während beim ursprünglichen Standard der antwortende Knoten seine Antwort direkt an sein eigenes ACK-Telegramm hängen konnte, so sind nun alle Knoten in der Lage, ein beliebiges asynchrones Telegramm an das ACK-Telegramm anzuhängen, falls das ACK-Telegramm an einem lokalen Child-Port empfangen wurde und das eigene Telegramm an den selben Knoten adressiert ist. Da der LLC nicht zwischen Daten mit und ohne GAP unterscheiden kann, ergibt sich dadurch lediglich eine zeitliche Optimierung ohne weitere Nachteile. Die einzige Bedingung bleibt weiterhin die Einhaltung des Fairness-Intervalls.

Im Original-Standard wurde das Konstrukt des „Fairness-Intervalls“ verwendet, um den Zugriff einzelner Knoten auf den Bus in der Weise zu limitieren, dass ein Knoten erst dann sein zweites Telegramm absetzen durfte, wenn alle vorhandenen Knoten zumindest die Gelegenheit hatten, ein erstes Telegramm abzusetzen. Dadurch, dass sich diese Regelung auf Anfragen *und* Antworten bezog, ergab sich daraus in Einzelfällen eine Benachteiligung derjenigen Knoten, die auf eine Antwort eines Knotens warteten, der bereits selbst ein erstes Telegramm abgesetzt hatte. Dieser musste erst das Fairness-Intervall abwarten, bevor er die Antwort schicken durfte. Der Mechanismus „**Priority Arbitration**“ ermöglicht nun eine priorisierte Arbitrierung (ohne Berücksichtigung des Fairness-Intervalls) für Knoten, die eine Antwort senden möchten.

### IEEE 1394b

Der b-Standard spezifiziert eine Reihe zusätzlicher Medien (POF bis $50m$ und GOF bis $100m$ Kabellänge) und auch ein abweichendes Kodierungsschema (8B/10B), welche nur für b-Standard-taugliche Geräte nutzbar sind. Zusätzlich wird eine Erhöhung der Datenrate und der möglichen Knoten-Entfernungen erzielt. Die Abwärtskompatibilität zum Betrieb von Geräten und Komponenten, die dem Standard 1394-1995 und 1394a-2000 entsprechen, ist weiterhin gegeben.

Eine Arbitrierung wurde bisher unter Berücksichtigung von GAP-Zeiten erreicht, die abhängig von PHY-Repeater-Verzögerungen, Kabellängen und der Anzahl von Knotenpunkten waren, aber unabhängig von der aktuellen Übertragungsgeschwindigkeit. Dadurch führte auch eine gesteigerte Bit-Übertragungsrate nicht zu einer entsprechenden proportionalen Erhöhung des Datendurchsatzes. Während beispielsweise bei der Verwendung von S400 ($400 MBit/s$) große Pakete immer noch eine Bus-Effizienz von 50% erreichten, lag die Effizienz bei kleinen Paketen (z.B. 64 Byte) nur bei 10%. Die Erweiterungen aus dem a-Standard griffen hier bereits konstruktiv ein, so dass eine erste Steigerung der Bus-Effizienz erzielt werden konnte, ohne jedoch die Nachteile der GAPs gänzlich zu eliminieren.

Nach wie vor war zwischen isochronem und asynchronem Transfer ein Subaction-GAP von $10\mu s$ notwendig. Auch im Anschluss an asynchrone Broadcasttelegramme (ohne anschließendes ACK-Telegramm) musste ein Subaction-GAP eingehalten werden. Für die Respektierung des Fairness-Intervalls war die Berücksichtigung des Arbitration-Reset-GAPs von $20\mu s$ für Viel-Sender erforderlich. Mit wachsender Ausdehnung des Kommunikationssystems stiegen auch die notwendigen Zeiten für die Signalisierungen von Arbitrierungsanfragen und -bestätigungen an, so dass bei wachsender Übertragungsrate die Effizienz litt.

Die **BOSS[10]-Arbitrierung** nutzt die Vollduplex-Eigenschaft der 1394b-Standard-Signalisierung. Bei ihrer Anwendung überschneiden sich die normale Datenkommunikation und die Arbitrierungssignalisierung auf dem Vollduplex-Bus, so dass die Anfragen der Knoten für beide Transfers (isochron und asynchron) innerhalb von „Pipelines“ für das aktuelle isochrone Intervall bzw. das Fairness-Intervall aufgenommen werden. Der „BOSS“ ist stets derjenige Knoten, der zuletzt (oder aktuell) ein gerichtetes asynchrones Paket versendet, da er der einzige Knoten ist, der an allen aktiven Ports Arbitrierungs-Anfragen empfangen kann und somit die Verantwortung für die nächste Arbitrierungsentscheidung am sinnvollsten tragen kann. Für weitere Signalisierungen, die in den Vorgängerstandards über GAP-Zeiten angezeigt wurden, verwendet BOSS unterschiedliche Token.

### IEEE 1394c

Im Zuge der Erschließung der Ethernet-Kommunikation erweitert der Standard IEEE 1394c die Funktionalitäten des PHY zur Ermöglichung des Transports von S800-Telegrammen ($800 Mbit/s$) des 1394b-Standards über CAT-5-Kabel (T-mode). Dadurch werden Steuerungssignale, Daten und Arbitrierung auf die Technologie 1000-BaseT ausgeweitet. Die Reichweite über CAT-5 UTP beträgt dabei $100m$. Unter Nutzung des gleichen Pinnings wie Ethernet auf RJ-45 bietet der neue Standard auch Funktionalitäten zur Aushandlung des verwendeten Protokolls, so dass Geräte damit entscheiden können, welches Protokoll (10BaseT Ethernet, 100BaseTx Ethernet, S100 1394b, 1000BaseT Ethernet, S800 1394c) sie verwenden wollen. Dabei tritt kein Konflikt mit den Geräten auf, die am gleichen Bus über den Standard IEEE 802.3 (Ethernet) kommunizieren.

[10] BOSS = Bus Owner/Supervisor/Selector

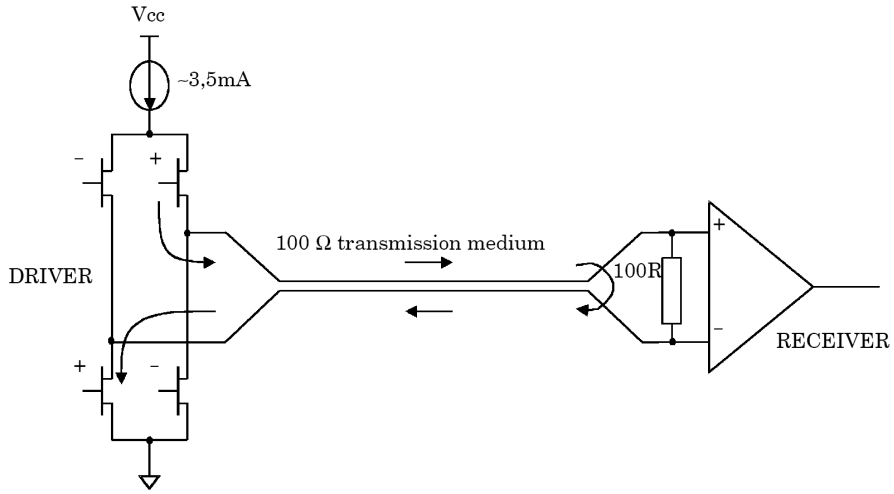



Abbildung 3.8: Treiber für LVDS-Leitungen

### 3.3.2 SpaceWire

Ursprünglich wurde SpaceWire zum Einsatz als Kommunikationssystem für die Raumfahrt entwickelt [79]. Es sorgt für eine Hochgeschwindigkeits-Datenverbindung zwischen Sensoren, Recheneinheiten, Massenspeicher und Subsystemen zur Übertragung von Informationen von und zur Erde. Zwei Standards, die die Spezifikation maßgeblich beeinflusst haben, sind IEEE 1355-1995 [80] und IEEE 1596.3-1996 [81] [82].

SpaceWire ist eine vollduplex, bidirektionale, serielle, Punkt-zu-Punkt Datenverbindung, die für jede Übertragungsrichtung zwei Signalleitungspaare verwendet, auf denen Signale nach der LVDS[11]-Spezifikation übertragen werden. Damit werden alle Teilnehmer über jeweils insgesamt acht Signalleitungen miteinander verbunden. Die Verbindungsleitungen zwischen zwei Teilnehmern sind maximal 10 m lang; Komponenten auf einem PCB[12] können ebenfalls über SpaceWire verbunden werden. Abbildung 3.8 zeigt einen Signalleitungstreiber für LVDS. Der konstante Strom von $3,5mA$ wird durch die Ansteuerung jeweils zweier versetzt gegenüber angeordneter Transistorpaare auf die Leitung gegeben. Je nach der sich ausprägenden Stromrichtung fällt dadurch am Empfänger eine Spannung von $\pm 350mV$ über dem abschließenden $100\Omega$-Widerstand ab, die hochohmig abgegriffen wird. Durch die Verwendung von Data-Strobe-Kodierung bei der Bitübertragung, die auch von FireWire (s.o.) genutzt wird, ist der Empfänger in der Lage, auch das Taktsignal des Senders zu gewinnen und so die vom Sender verwendete Übertragungsgeschwindigkeit nach bereits einem Bit selbst zu erkennen. Dabei sind Übertragungsraten von bis zu $200MBit/s$ realisiert.

SpaceWire wird in vielen Raumfahrtprojekten der ESA, NASA und JAXA erfolgreich eingesetzt. Ein Einsatz innerhalb der Komponenten eines Robotersystems, wie sie im SFB 562 verwendet werden, ist denkbar, da die favorisierte Systemstruktur ähnlich ist (PCI-Karte im Standard-PC und verteilte Komponenten). Die Implementierung

[11] LVDS = low voltage differential signal
[12] PCB = printed circuit board, Platine

von SpaceWire für die verteilten Komponenten erfolgt dabei in ASICs oder FPGAs und kann auf diese Weise leicht mit weiteren bereits entwickelten Funktionen kombiniert werden. PCI-Steckkarten sind zur Nutzung mit Standard-PC erhältlich [83]. Allerdings existieren Treiber bisher für die Betriebssysteme Windows und Linux. Im Gegensatz zum Physical Layer von FireWire wird dieser bei SpaceWire nicht weiter entwickelt, so dass die maximale Übertragungsrate von $200 MBit/s$ als absolute Grenze gesehen werden muss.

# 4 Kommunikations-Infrastruktur

In diesem Kapitel erfolgt die Beschreibung des Konzepts und der Realisierung einer speziell für den Anwendungsbereich in hochdynamischen Robotersystemen entworfenen Kommunikationsinfrastruktur, die am Institut für Elektrische Messtechnik und Grundlagen der Elektrotechnik der TU Braunschweig im Rahmen des Sonderforschungsbereichs 562 „Robotersysteme für Handhabung und Montage“ entwickelt wurde. Sie wird in diesem Projekt auch zukünftig als Kommunikationsgrundlage für Funktionalitäten aus der Regelungs- und Steuerungstechnik verwendet, um die Entwicklung und kooperative Anwendung modularer, parallelroboterspezifischer Soft- und Hardwarekomponenten zu unterstützen und im Echtzeitkontext durchzuführen.

In Kapitel 2.3 wurde erläutert, aus welchen Gründen die Anwendung von Middlewaresystemen generell sinnvoll und empfehlenswert ist. Gerade für die Anwendung innerhalb eines Sonderforschungsbereichs, in dem unterschiedliche Komponenten in Form von Soft- und Hardwaremodulen realisiert werden und in einem Gesamtkontext kooperieren müssen, bietet es sich an, middlewarebasierte Kommunikationsmechanismen zu verwenden, um klare Modulschnittstellen definieren und zeitkritische Abläufe für den Modulentwickler transparent halten zu können. Letztlich lassen sich somit alle Funktionalitäten dynamisch in Form eines Baukastensystems zusammenstecken und mittels einer übersichtlichen Konfigurationsdatei für die Laufzeitanwendung konfigurieren.

Soll beispielsweise ein neues Regelungskonzept zur Anwendung kommen, eine defekte Soft- oder Hardwarekomponente durch eine Alternative ersetzt werden oder beim Erreichen eines vorgegebenen Anwendungs- oder Systemzustands eine komplett neue Hardware an die verarbeitenden Prozesse angebunden und informationstechnisch integriert werden, so kann dies unmerklich für den Benutzer geschehen, wenn entsprechende organisations- und kommunikationstechnische Mechanismen bereitgestellt und verwendet werden. Dabei ist der Austausch von Soft- und Hardwarekomponenten auch zur Laufzeit des Roboters ohne Neu-Kompilierung des Systems möglich, und alle neuen Variablen sind in allen Softwareschichten zugänglich.

Aber auch für die Entwicklung der Kommunikationsinfrastruktur selbst besteht der Anspruch, die internen Funktionalitäten möglichst aus austauschbaren Modulen mit einheitlichen Schnittstellen aufzubauen, damit Modifikationen und Erweiterungen übersichtlich und ohne Querbeeinflussung anderer Funktionen erfolgen können.

Die Hauptkomponenten der in dieser Arbeit beschriebenen Infrastruktur sind das IEEE1394-Kommunikationsmedium (FireWire), das darauf aufsetzende deterministische Industrial Automation Protocol (IAP) und die Echtzeitmiddleware MiRPA-X. Abbildung 4.1 zeigt die zur Kommunikationsinfrastruktur gehörigen Soft- und Hard-

waremodule und gibt mit der Zuordnung zu verwendeten Rechnerplattformen auch eine Übersicht über den Umfang und die Aufteilung der entwickelten Anwendung aus kommunikationstechnischer Sicht.

Es sind hier fünf Rechnersysteme (ein Steuerungs-PC, zwei Remote-PC und zwei DSP-Systeme) dargestellt, die über jeweils spezifische Kommunikationshardware (hier als dunkelgraue Blöcke dargestellt) miteinander verbunden sind. Die hellgrauen Blöcke kapseln die jeweiligen Softwarefunktionalitäten, während die weiß hinterlegten Blöcke die Programmierschnittstellen darstellen, die von angrenzenden Softwaremodulen zum Zugriff auf Funktionalitäten verwendet werden.

Auf dem zentralen Steuerungs-PC werden innerhalb der Applikations-Server und Applikations-Clients alle benötigten Funktionalitäten zur Berechnung der Sollgrößen für die Aktoren und Antriebe zur Verfügung gestellt. Diese kommunizieren zunächst nicht direkt miteinander, sondern verwenden die von der Middleware angebotenen Kommunikationsmechanismen, um Daten auszutauschen und über Schnittstellen nach außen zu kommunizieren. Diese Schnittstellen sind in Form von Softwaremodulen (TCP-Server und IAP-Master) realisiert, die ebenfalls über die Mechanismen der Middleware am lokalen Kommunikationsgeschehen teilnehmen, aber nach außen hin direkt auf die entsprechende Kommunikationshardware (Ethernet oder FireWire) zugreifen. Die PC im oberen Teil von Abbildung 4.1 sind über Ethernet und TCP/IP mit dem zentralen Steuerungsrechner verbunden. Die dort angesiedelten Softwaremodule bearbeiten für die Echtzeitregelung zeitlich unkritische Funktionen, wie z.B. Visualisierungen oder neue Sollwertgenerierungen auf oberer Applikationsebene. Auf der rechten Seite befinden sich die Recheneinheiten der untersten Hierarchieebene. Die sind als Digitale-Signalprozessorsysteme ausgeführt und mittels des IEEE1394-Standards am zentralen Steuerungsrechner angeschlossen. Die hier laufenden Applikationen sorgen für die Kommunikationsorganisation (IAP-Node), die Echtzeitsynchronisation (CycleTimer) und die applikationsspezifische Anbindung an die Roboterhardware wie Antriebe, Sensoren und allgemeine Aktuatoren.

## 4.1 Anforderungen an die Entwicklung

In Kapitel 1 wurden allgemeine Anforderungen an eine für parallele Robotersysteme innerhalb des SFB 562 speziell nutzbare Kommunikationsinfrastruktur vorgestellt. In diesem Abschnitt folgen weitere Anforderungen, die als Richtlinien für Dimensionierung und Implementierung von Kommunikationsfunktionalitäten herangezogen werden sollen.

### Grundsätzliches zur Implementierung von Software

Zur Implementierung von Software wurde im SFB 562 die prozedurale Programmiersprache C ausgewählt, da sie für alle im Projekt verwendeten Computersysteme (PC, spezielle Mikrocontroller, DSP) verfügbar ist und auch in der einfachen nicht-objektorientierten Form fachgebietübergreifend den größten Bekanntheitsgrad

für hardwarenahe Programmierung aufweist. Die verfügbaren Entwicklungsumgebungen sind für jedes Betriebssystem so weit ausgereift, dass bei der Softwareentwicklung auf komfortable Testfunktionalitäten zurückgegriffen werden kann. Ein wesentlicher Vorteil von C-Programmierung liegt auch darin begründet, dass automatische Mechanismen (z.B. Garbage Collection, dynamisches Binden uvm.) zur Laufzeit nicht existieren und somit der Programmierer sehr große Kontrolle über das Laufzeitverhalten seiner Implementierungen hat. Eine nicht zu verachtende Herausforderung liegt allerdings darin, bei der Vielzahl der zu realisierenden Teilkomponenten nicht den Überblick über implementierungsspezifische Details zu verlieren. Besonders für noch folgende Weiterentwicklungen durch Mitarbeiter und Studenten im SFB wurde daher ein Merkblatt erstellt, mit dessen Hilfe sowohl das Zurechtfinden in den bestehenden Implementierungen als auch das Einpflegen neuer Funktionalitäten nach bewährtem Muster ermöglicht wird.



Da ein wesentlicher Teil der praktischen Arbeit der Umsetzung der entwickelten Konzepte diente, wird in diesem Abschnitt detailliert auf die für die programmiertechnische Realisierung hilfreichen Richtlinien eingegangen:

**Realisierung von Softwareebenen** Um die Komplexität des zu entwickelnden Systems beherrschbar zu machen, ist es erforderlich, die zugehörige Software in Ebenen (Schichten) zu organisieren, die klar voneinander (über eindeutig definierte Schnittstellen) getrennt sind und sich jeweils in dem Grad der Hardwarenähe unterscheiden. Da die konsequente Umsetzung dieser Art von Schichtung zu einem Leistungsverlust bei der Ausführung der Software führen kann (durch Kopiervorgänge zwischen den Schichten), ist im Einzelfall zu prüfen, ob sie in der Implementierung teilweise zugunsten der Ausführungsgeschwindigkeit aufgehoben werden sollte.

**Aufteilung in Module** Um dem Objekt-Gedanken trotz Verwendung einer prozeduralen Programmiersprache nachzukommen, werden die Implementierungen abgrenzbarer Funktionseinheiten *innerhalb* einer Softwareschicht (s.o.) in unterschiedlichen Modulen gekapselt. Die Module tragen einen kurzen Namen, der die inhaltliche Einordnung der Modulfunktionalitäten eindeutig ermöglicht. Ein Zugriff auf modulinterne Zustandsvariablen ist nur über moduleigene Funktionsaufrufe möglich.

**Verwendung von Dateien** Um den Programmcode übersichtlich zu halten und die verwendete Struktur auch in den angelegten Quelldateien sichtbar werden zu lassen, ist zu jedem Modul eine Definitions- und eine Quellcodedatei mit der Namensbezeichnung des Moduls zu erstellen. Öffentliche, modulbezogene Funktionsdeklarationen und entsprechende Typen- und Konstantendefinitionen sollen in den Definitionsdateien (.h), dagegen modulprivate Funktionsdeklarationen, Typen- und Konstantendefinitionen sowie Funktions- und Variablendefinitionen in den Quellcodedateien (.c) angelegt werden. Diese Verwendung begünstigt die objektorientierte Denkweise, lässt eine einfache Realisierung von Zugriffsbeschränkungen zur Kompilierzeit zu und führt zu übersichtlichem Programmcode.

**Programmierschnittstellen** Für jedes abgeschlossene Softwareprojekt muss eine Pro-

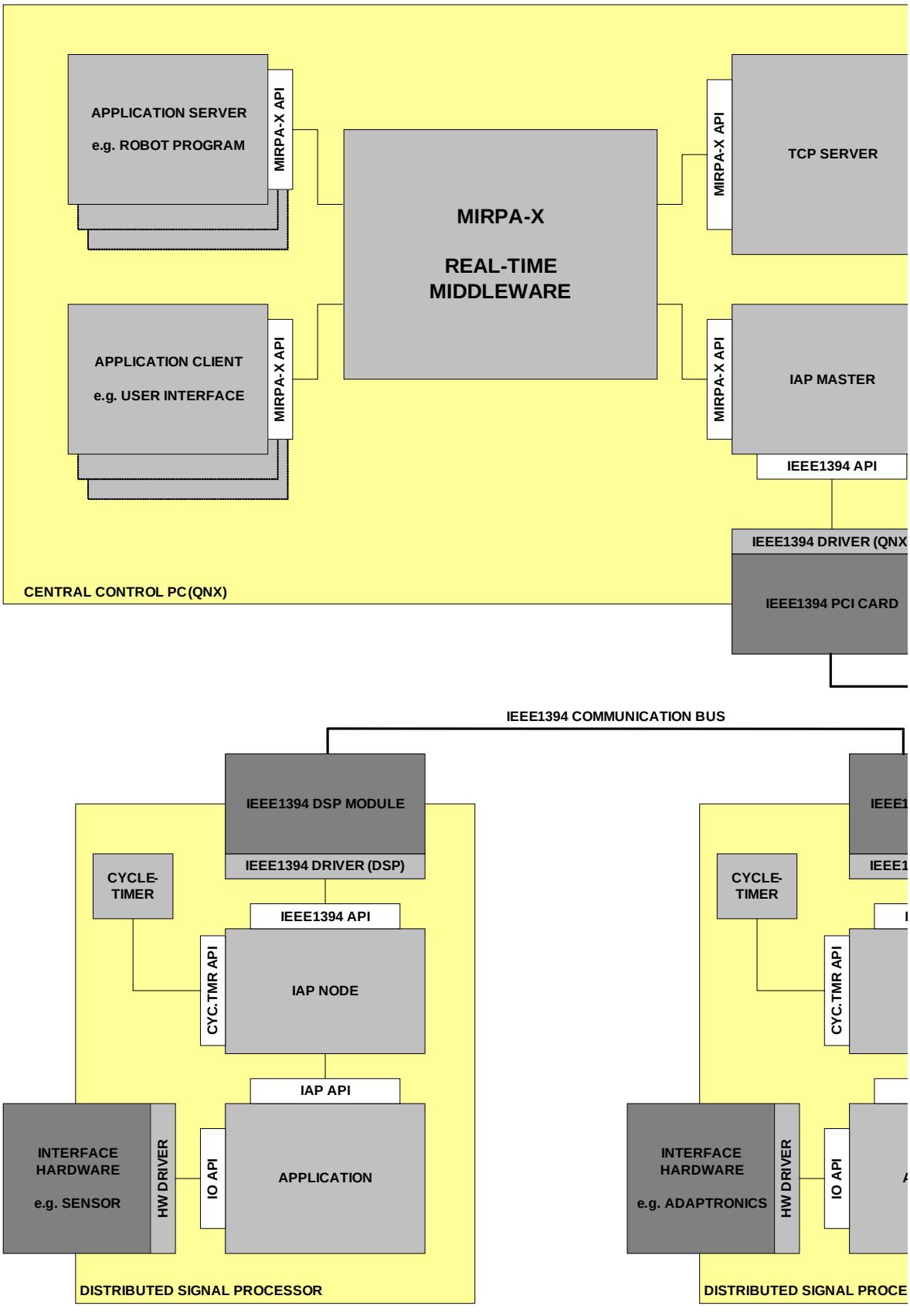
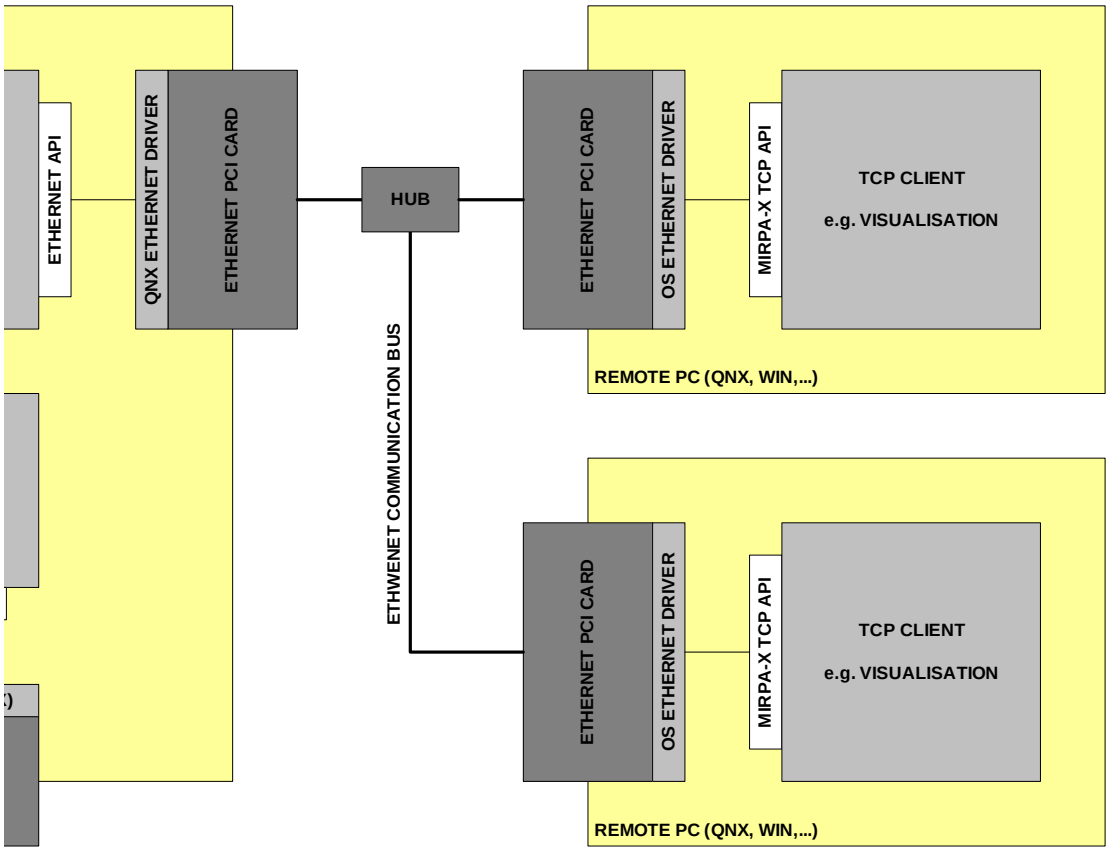
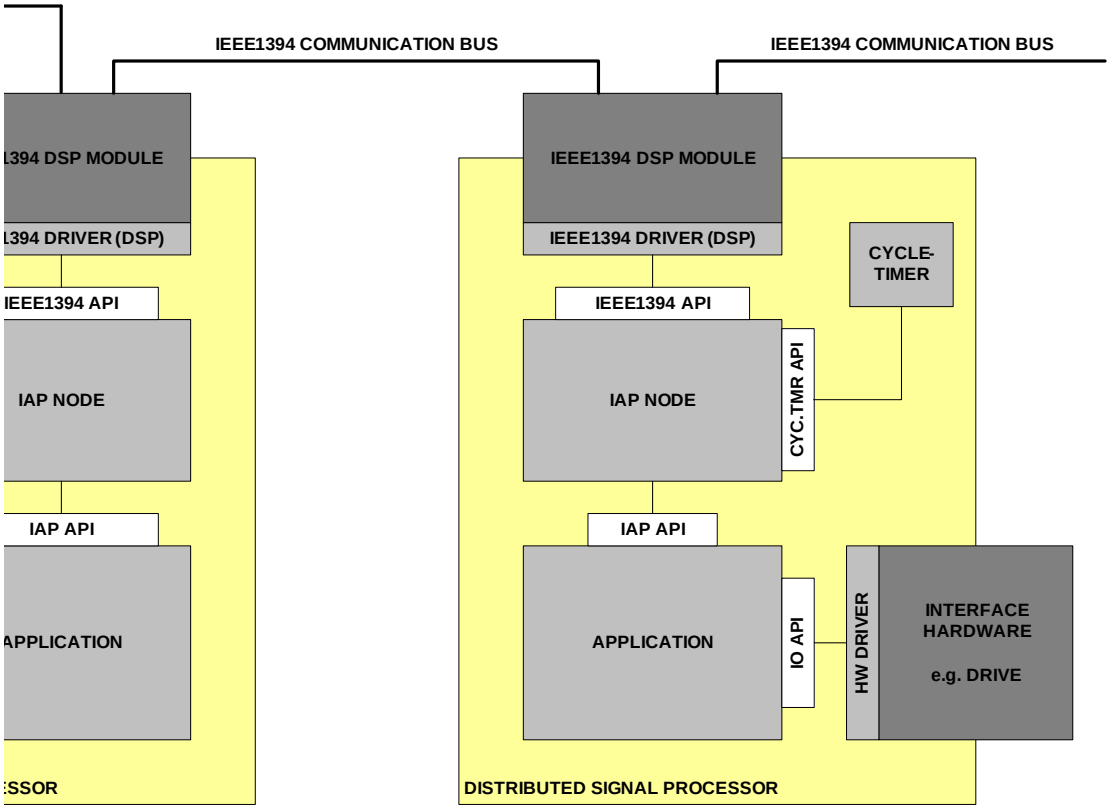



Abbildung 4.1: Soft- und Hardwaremodule der Kommunikations-Infrastruktur


ETHERNET API
QNX ETHERNET DRIVER
ETHERNET PCI CARD
HUB
ETHERNET PCI CARD
OS ETHERNET DRIVER
MIRPA-X TCP API
TCP CLIENT
e.g. VISUALISATION
REMOTE PC (QNX, WIN,...)
ETHWENET COMMUNICATION BUS
ETHERNET PCI CARD
OS ETHERNET DRIVER
MIRPA-X TCP API
TCP CLIENT
e.g. VISUALISATION
REMOTE PC (QNX, WIN,...)


IEEE1394 COMMUNICATION BUS
IEEE1394 COMMUNICATION BUS
IEEE1394 DSP MODULE
IEEE1394 DRIVER (DSP)
IEEE1394 API
IAP NODE
IAP API
APPLICATION
IEEE1394 DSP MODULE
IEEE1394 DRIVER (DSP)
CYCLE-
TIMER
IEEE1394 API
IAP NODE
CYC.TMR API
IAP API
APPLICATION
IO API
HW DRIVER
INTERFACE
HARDWARE
e.g. DRIVE
DISTRIBUTED SIGNAL PROCESSOR

grammier- oder Benutzerschnittstelle zur Verfügung gestellt werden, über die ein Zugriff aus anderen Softwareebenen oder -modulen auf die implementierten Funktionalitäten ermöglicht wird. Diese Schnittstelle (API) wird pro Softwareprojekt jeweils über eine kompilierte Bibliotheksdatei (library) und eine Definitionsdatei (header) veröffentlicht. Aber auch innerhalb von Softwareprojekten ist es erforderlich, Module eindeutig gegeneinander abzugrenzen (s.o.), auch wenn diese nicht in Form einer Bibliothek vorliegen müssen.

**Kommentare mit DoxyGen** Innerhalb des C-Quellcodes werden die implementierungsbezogenen Kommentare sowie die Funktionsbeschreibungen für Benutzer in einem einheitlichen Format erstellt. Dabei wird das Tool „DoxyGen“ verwendet, das Kommentarfunktionalitäten in einer HTML-ähnlichen Form zur Verfügung stellt. Auf diese Weise lassen sich Beschreibungen für die entwickelten Module und Projekte automatisch strukturiert und einheitlich dargestellt erzeugen.

**Variablen- und Funktionsnamen** Der Name der Funktion innerhalb eines Moduls ergibt sich aus dem jeweiligen Modulnamen, gefolgt von einem Unterstrich und einer Repräsentation der implementierten Funktionalität im englischen oder deutschen Klartext. Dabei beginnt jedes neue Wort mit einem Großbuchstaben (z.B. bezeichnet `Config_GetPresentStatus()` eine Funktion, die den aktuellen Status innerhalb des Konfigurationsmoduls zurück gibt). Die Variablen erhalten ebenfalls einen Namen im Klartext, jedoch angeführt von einem entsprechenden Typenbezeichner (z.B. `acModuleName` für ein Feld (**a**rray) bestehend aus Zeichen (**c**har), das den Modulnamen enthält). Auch die Dateinamen ergeben sich eindeutig aus der Modulzugehörigkeit.

Zur gesamtheitlichen Dokumentation nicht des Quellcodes direkt, sondern der strukturellen und funktionalen Zusammenhänge wird im SFB 562 einheitlich die UML verwendet [151]. Diese weit verbreitete Sprach- und Modellierungsspezifikation wird durch eine darauf aufbauende Spezifikation für „Schedulability, Performance, and Time“ erweitert [152] und kann damit nun zur Dokumentation von strengen zeitlichen Zusammenhängen und Echtzeitaspekten verwendet werden. In [153] wird ein Vorschlag erarbeitet, um mittels gezielter Anwendung der erwähnten Spezifikation auch die Leistungsgrenzen des Systems schon während der Entwicklung überprüfen zu können.

Besondere Aufmerksamkeit gilt, neben diesen eher die Quellcodeverständlichkeit und -modularität fördernden Vereinbarungen, der Bereitstellung einer hohen Verfügbarkeit (high availability) während der Laufzeit der Software. In diesem Zusammenhang ist es notwendig, durch Nutzen von fehlererkennenden und -behebenden Betriebssystemfunktionen, die mittlere Zeit zwischen zwei aufeinander folgenden Softwarefehlern (mean time between failures, MTBF) zu maximieren, während die Zeit zum Wiederaufstart des Computersystems (mean time to recover, MTTR) minimiert werden soll.

In den folgenden Abschnitten werden die einzelnen realisierten Komponenten zunächst grundsätzlich in ihren Eigenschaften und anschließend im detaillierten Aufbau vorgestellt.

## 4.2 Middleware MiRPA-X

Der Einsatz einer Middleware zur Organisation von Kommunikationsvorgängen zwischen unterschiedlichen Applikationsmodulen auch auf verschiedenen Rechnern ist besonders sinnvoll, wenn es darum geht, voneinander abgrenzbare (modulare) Systemteile kooperierend arbeiten zu lassen. Im Sonderforschungsbereich 562 werden nicht nur sehr unterschiedliche Softwaremodule entwickelt, sie werden auch noch von unterschiedlichen Personen implementiert, benötigen unterschiedliche Kommunikationsmechanismen und unterliegen ständigen Änderungen. Die Organisation des Nachrichtentransfers und der Nachrichtentransfer selbst müssen den in den Abschnitten 2 und 3 beschriebenen Anforderungen zu jeder Zeit genügen, um einen zuverlässigen Hochgeschwindigkeits-Echtzeitbetrieb eines Steuerungssystems zu ermöglichen. Die Erläuterungen zu den Softwaretechnologien aus den Abschnitten 2.3 und 2.4 zeigen dabei, welche Leistungsfähigkeit heutige Kommunikationsmechanismen aufweisen und wo ihre Schwachstellen bezüglich realisierbarer Zyklus- und Latenzzeiten liegen.

### 4.2.1 Funktionsweise MiRPA-X

In 2.3.2 wurde die Middleware „MiRPA“ vorgestellt, die am Institut für Robotik und Prozessinformatik der TU Braunschweig entwickelt wurde. Die Leistungsfähigkeit dieser Implementierung entsprach zum Zeitpunkt der Arbeiten innerhalb des Sonderforschungsbereichs 562 nicht dessen deterministischen Anforderungen bzw. konnte die Problematik der Prozess-Synchronisation nicht zufriedenstellend lösen. Das Zusammenführen unterschiedlicher Zielvorstellungen im Bereich einer Middleware für Robotersteuerungen konnte leider nicht erfolgen, so dass im Rahmen dieser Arbeit die Middleware „MiRPA-X“ neu und mit neuartigen Funktionalitäten als Kern einer Kommunikationsinfrastruktur implementiert wurde. Zusammengefasst ergeben sich für MiRPA und MiRPA-X folgende Unterschiede und Anwendungsbereiche:

**MiRPA** Nachrichtenorientierter Kommunikations-Softwarebus zur Anwendung für die Steuerung serieller Roboterstrukturen mit realisierten Zykluszeiten im Bereich von wenigen Millisekunden.

**MiRPA-X** Herzstück einer echtzeitfähigen Kommunikationsinfrastruktur mit zyklischem Scheduling, Nachrichtenkommunikation, Shared-Memory-Kooperation und Prozesssynchronisation zur Anwendung für komplexe Parallelroboterstrukturen mit realisierbaren Zykluszeiten von bis zu $125\mu s$.

Im Zuge der Vorstellung des MiRPA-Ansatzes des iRP, wurden bereits einige Mechanismen erwähnt, die als essenzieller Teil *jeder* Middleware auch in MiRPA-X realisiert sind. In diesem Abschnitt erfolgt eine ausführliche Darstellung der wichtigsten Mechanismen und Funktionalitäten von MiRPA-X.

### Konzept

MiRPA-X ist eine Middleware, die einen vollständig transparenten Datenverkehr zwischen konstruktiv interagierenden Softwaremodulen ermöglicht, ohne dabei Dateninhalte zu berücksichtigen oder zu bewerten. Damit geht die Einsatzmöglichkeit von MiRPA-X über den Einsatzbereich innerhalb von Robotersteuerungen hinaus und kann damit auch in anderen Bereichen (z.B. Automobilkommunikation) erfolgen. Die Umsetzung des Nachrichtenmechanismus über ein konsequent implementiertes Client/Server-Modell für die über MiRPA-X verbundenen Softwaremodule garantiert grundsätzliche Deadlockfreiheit für die Kommunikation. Jeder beliebige Client-Prozess, der zu MiRPA-X eine Verbindung aufbaut, kann nachrichtenbasierte Daten- oder Dienstanfragen stellen, die daraufhin von MiRPA-X an die entsprechenden Dienstleister (Server) weitergeleitet werden.

MiRPA-X unterstützt das objektorientierte Design eines Anwendungssystems (z.B. Robotersteuerung) und ermöglicht den Anwendern auch einen objektorientierten Umgang mit den bereitgestellten Funktionalitäten dieses Systems:

- Nachrichtennamen entsprechen Objektreferenzen (DYNAMISCH)
- Objektschnittstelle wird statisch in einer ASCII-Datei festgelegt
- Nachrichtendaten realisieren Methodenaufrufe (DYNAMISCH)
- Fehlerrückgaben realisieren Ausnahmebehandlungen (DYNAMISCH)
- Garbage-Collection entfällt, da nicht notwendig/sinnvoll

Die Anwendung von MiRPA-X gestaltet sich in der Weise, dass Systemplaner benötigte Funktionalitäten modular gekapselt realisieren und über MiRPA-X (als Server) zur Verfügung stellen oder aber auf die zur Verfügung gestellten Funktionalitäten zugreifen (als Client). Für den Fall, dass mehrere Systemplaner gemeinsam beispielsweise ein Steuerungssystem aufbauen, können die verwendeten strukturierten Daten in einer allgemein zugänglichen Datei (gemeinsam mit einer Beschreibung) vereinbart werden. Diese Konfigurationsdatei wird zur Laufzeit von MiRPA-X eingelesen und dient als zusätzliche Überwachungshilfe für die Datenkommunikation über den ObjectServer. Zur Laufzeit anstehende Konfigurationsänderungen und allgemeine Informationsanfragen an den ObjectServer erlauben das nachträgliche Umstrukturieren des gesamten Steuerungssystems in Abhängigkeit beispielsweise von Applikationsereignissen.

Da MiRPA-X selbst unabhängig von externen Kommunikationssystemen ist, kann über spezielle Servermodule Rooterfunktionalität realisiert werden. In der vorliegenden Arbeit ist dieses über das IAP-Modul (s. Abschnitt 4.3) geschehen, um zu externen Roboterkomponenten (Sensoren und Aktoren) unter Nutzung des IEEE1394-Standard Echtzeitkommunikation zu ermöglichen. Über ein weiteres Ethernet-Servermodul sind auch mehrere Windows-PC (allerdings ohne Echtzeitanspruch) in das Steuerungssystem integriert worden.

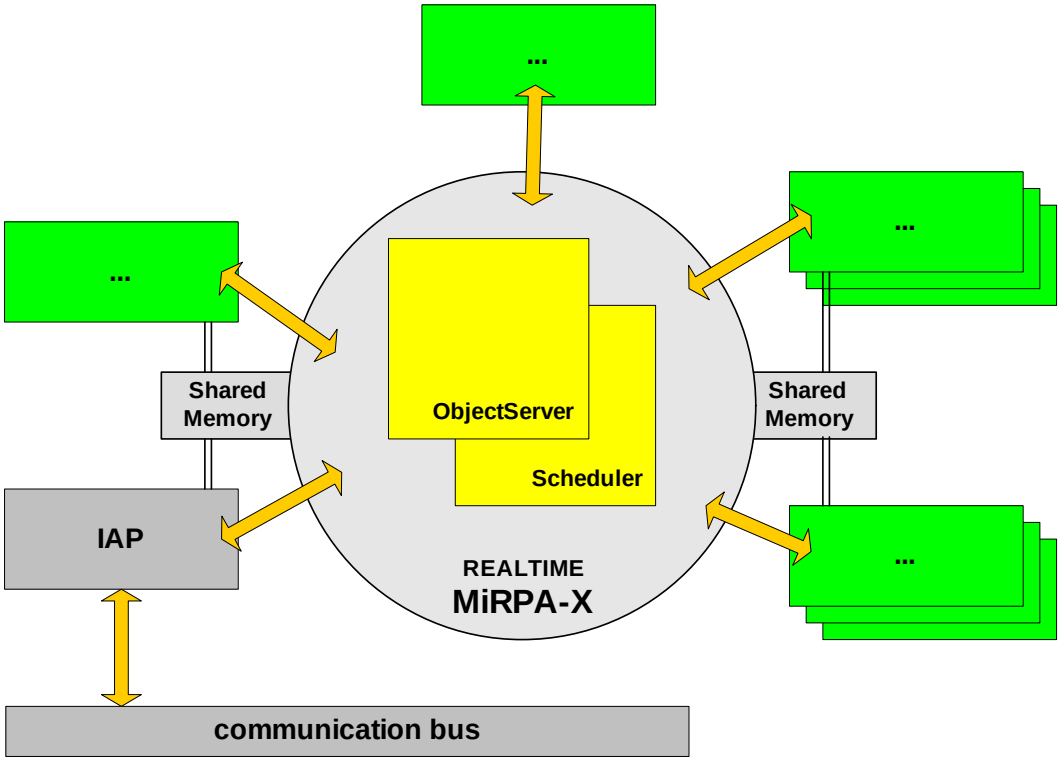



Abbildung 4.2: Anwendungsstruktur von MiRPA-X

Abbildung 4.2 zeigt einen schematischen Überblick über die modulare Anwendungsstruktur von MiRPA-X.

Neben den als dunkle Blöcke dargestellten Applikationsmodulen (diese können sich zur Laufzeit auf beliebigen Steuerungsebenen befinden) existieren auch zwei spezielle Softwaremodule in der Abbildung, nämlich „IAP“ und „communication bus“, auf die in den Abschnitten 4.3 und 4.4 eingegangen wird. Sie realisieren gemeinsam eine Kommunikationsfunktionalität als Schnittstelle zu den externen Komponenten des Steuerungssystems. Die allgemeinen Applikationsmodule können unter einander über die von MiRPA-X bereitgestellten Kommunikationsfunktionalitäten Daten austauschen. Diese Funktionalitäten basieren generell auf zwei Mechanismen: Nachrichten-Kommunikation (Verbindungspfeile) und Shared-Memory-Kooperation (Verbindungslinien). Während die Nachrichtenkommunikation auf den blockierenden Nachrichtenfunktionalitäten des unterlagerten Echtzeitbetriebssystems beruhen, stützt sich die Kooperation über Shared-Memory auf keine explizite Betriebssystemfunktionalität und erfordert somit die Implementierung eigener Sicherheitsmaßnahmen, die später beschrieben werden.

Neben der strukturunterstützenden Funktion stellt MiRPA-X auch Funktionalitäten zur zeitlichen Strukturierung von Abläufen innerhalb der Anwendung zur Verfügung. Dazu dient eine Ablaufsteuerung, die im MiRPA-X-Kontext als *Scheduler* bezeichnet wird, sich aber deutlich von dem Standard-Scheduler des unterlagerten Betriebssystems unterscheidet.

Im Folgenden sollen die hier erwähnten Grundfunktionen näher erläutert werden.

### Nachrichtenbasierte Kommunikation

MiRPA-X nutzt die Inter-Prozess-Kommunikation (IPC) des unterlagerten Betriebssystems (hier QNX), um die synchrone Nachrichtenkommunikation zu realisieren. Synchron bedeutet in diesem Zusammenhang, dass der empfangende Kommunikationspartner (Server) *blockiert* auf das Eintreffen der Nachricht wartet und seine Arbeit aufnimmt, sobald die erwartete Nachricht eintrifft. Im Allgemeinen erfolgt nach der serverseitigen Datenauswertung und Berechnung eine Bestätigung an den beauftragenden Prozess (Client).

Die gesamte nachrichtenbasierte Kommunikation wird von dem *ObjectServer*, der Kernkomponente von MiRPA-X, organisiert und durchgeführt. Dieser wirkt als eigenständiger Thread innerhalb der Steuerungsanwendung, der Clientanfragen auswertet und bei Bedarf an entsprechende Serverprozesse weiterleitet. Die Strategie zum Weiterleiten basiert auf einer zwischen Client und Server einheitlichen Namenskonvention für die jeweiligen Applikationsdienste (z.B. „EVAL_NEW_SKILL_PRIMITIVE“). Dabei wird jeder Dienst intern über eine eindeutige Referenz (ein Hashwert) identifiziert, um das Auffinden von Zielservern im ObjectServer zu beschleunigen. Der Einsatz eines solchen Namensdienstes ermöglicht einen für die Anwendung transparenten und flexiblen Weg, modulare und sogar redundante Applikationsdienste zu implementieren, ohne die Identität und den aktuellen Aufenthaltsort des dazugehörigen Servers auf der Applikationsebene zu kennen.

In Abbildung 4.3 sind die grundsätzliche Anwendungsstruktur für den ObjectServer und die zur Erfüllung der Vermittlungs- und Verwaltungsaktivität verwendeten Nachrichtentypen dargestellt. Sie werden im Folgenden erklärt.

Um einen neuen Dienst im System zu registrieren, versendet der entsprechende Serverprozess eine Konfigurationsanfrage, die den gewünschten Dienstnamen und Diensttyp enthält. In Abhängigkeit von dem angefragten Diensttyp prüft der ObjectServer nach Kollisionen mit früheren Registrierungen für diesen Dienst und führt die Registrierung durch. Sobald auf diese Weise die einzelnen Applikationsdienste des Systems registriert wurden, können nun Anfragen der Clientprozesse positiv erfüllt werden. Innerhalb von MiRPA-X werden dabei drei grundsätzliche Arten von Anfragen unterschieden:

**COMMAND** Dieser Anfragetyp ist für den Clientprozess als nicht-blockierend implementiert. Der Client hat also die Möglichkeit (je nach Prioritätsbeziehung zum Server) nach Versand der Anfrage weiter zu arbeiten, während der zugehörige Server die empfangene Anfrage bearbeitet. Es ist möglich, eine beliebige Anzahl von Servern für einen COMMAND zu registrieren. Alle für diese Nachricht registrierten Server erhalten die Anfrage (Publish/Subscribe).

**REQUEST** Dieser Anfragetyp lässt den anfragenden Client so lange blockiert, bis der entsprechende Server die Anfrage bearbeitet und beantwortet hat. Die Antwort

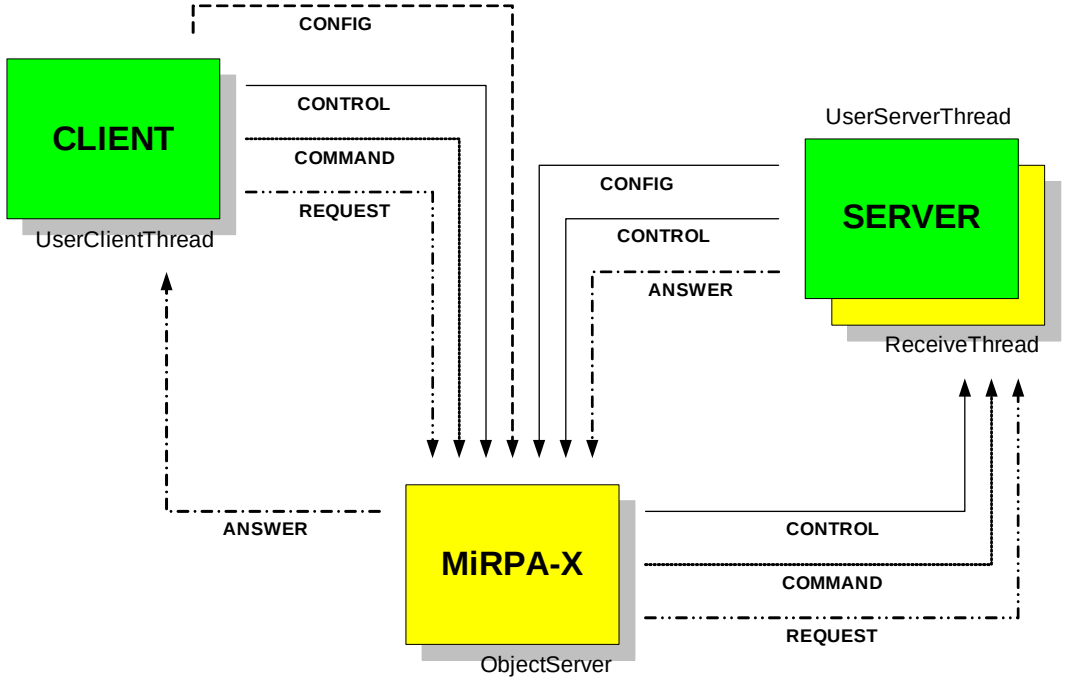



Abbildung 4.3: Nachrichtentypen zur Kommunikation mit dem ObjectServer

wird danach vom Client weiter ausgewertet. Es besteht hier die Möglichkeit, einen oder mehrere redundante Server zu verwenden, wobei in letzterem Fall alle weiteren Antworten nach der ersten verworfen werden.

**ANSWER** Dieser Nachrichtentyp wird genutzt, um Antworten auf Anfragen des Typs REQUEST zu versenden. Er stellt den einzigen Nachrichtentyp dar, den ein Serverprozess selbständig initiieren kann. Empfängt der ObjectServer eine Nachricht dieses Typs, entscheidet er anhand interner Aktivitätslisten, ob ein auf den Empfang blockierter Empfänger (Client) zugeordnet werden kann und leitet die ANSWER-Nachricht weiter.

MiRPA-X realisiert auf der API-Ebene jedes laufenden Serverprozesses einen eigenen (für den Nutzer unsichtbaren) Thread, der vom ObjectServer ankommende Nachrichten empfängt, bestätigt und an den Nutzer auf Serverseite weitergibt (Mailbox, s. Abschnitt 2.1.2). Damit ist sichergestellt, dass der ObjectServer auch bei Fehlverhalten eines Servermoduls nicht blockiert werden kann und darum weiterhin für Anfragen anderer Applikationsteile empfangsbereit bleibt. Die Nachrichtenübermittlung an den Server erfolgt dabei gemäß der Priorität des anfragenden Clients, während die Ausführung der gewünschten Funktionalität in Abhängigkeit von der Serverpriorität stattfindet.

Prinzipiell besteht die Möglichkeit, dass ein Serverprozess, sobald eine Nachricht empfangen wird, wieder eine Nachricht an einen anderen Serverprozess versendet und damit selbst auch zum Clientprozess wird. Die Implementierung von MiRPA-X in der aktuellen Version unterbindet allerdings solche kaskadierten Nachrichtenübertragungen, um eine für den Systementwickler übersichtliche Systemstruktur sicherzustellen.

Kommunikation zwischen Servern muss, wenn überhaupt erwünscht, über andere von MiRPA-X angebotene Kommunikationsmechanismen erfolgen.

Weitere in Abbildung 4.3 dargestellte Nachrichtentypen werden vor allem während der Konfigurations- oder Pausephasen des Robotersystems verwendet:

**CONFIG** Dieser Nachrichtentyp wird dazu verwendet, die Konfiguration des ObjectServers und aller anderen Komponenten von MiRPA-X vorzunehmen. Server und Client können diesen Typ gleichermaßen versenden. Auf diese Weise lassen sich beispielsweise die schon erwähnten Serverdienste anmelden, oder Einfluss auf das Echtzeitverhalten der Ablaufsteuerung von MiRPA-X ausüben.

**CONTROL** Um als Systementwickler eine Übersicht über das für die Laufzeit aufgebaute Kommunikationsnetz mit verfügbaren Diensten und aktiven Servern zu erhalten, kann dieser Nachrichtentyp verwendet werden. Der ObjectServer wird dazu veranlasst, bereits registrierte Dienste, registrierte gemeinsame Speicherbereiche und die aktuelle Aktivität der Echtzeit-Ablaufsteuerung auszugeben.

### Synchronisation und Kooperation

Neben der nachrichtenbasierten Kommunikation und der Ablaufsteuerung für Echtzeitprozesse stellt MiRPA-X weitere Mechanismen zur Verfügung, mit deren Hilfe Kooperation und Synchronisation von Applikationsprozessen möglich ist. Grundsätzlich beziehen sich diese Mechanismen auf die Organisation von über Prozessgrenzen hinweg nutzbare Speicherbereiche (Shared-Memory).

Ähnlich wie bei der Nachrichtenkommunikation, verwendet MiRPA-X zur Verwaltung von Shared-Memory einen Namensdienst. Dieser stellt die Modularität und Transparenz bei der Entwicklung von Applikationsmodulen sicher. Die Nutzung von Shared-Memory ermöglicht verzögerungsfreie, nicht-blockierende Kommunikation (Kooperation) unter Anwendungsprozessen. Dabei übernimmt MiRPA-X nur die Aufgabe der Verwaltung, nicht aber die der Durchführung von Shared-Memory-Zugriffen. Bei der Registrierung eines Shared-Memory-Bereiches erhält der registrierende Prozess eine lokale Referenznummer (Handle) und eine Speicheradresse, in dem die gewünschten Daten zu finden sind bzw. sein werden.

In dem Fall, dass mehrere Applikationsprozesse auf den gleichen SharedMemory-Bereich lesend oder schreibend zugreifen wollen, muss zu jeder Zugriffszeit Datenkonsistenz sichergestellt werden. Da aber die Nutzung von Shared-Memory nichtblockierend und generell unsynchronisiert ist und damit in sich keinen Schutzmechanismus für das Absichern des „kritischen Abschnitts“ (critical section, [5]) bereitstellt, sind weitere absichernde Mechanismen erforderlich, die auch in dieser Arbeit implementiert wurden. Die folgende Aufstellung gibt eine Übersicht über die den Shared-Memory-Mechanismus sicherstellende Funktionalitäten und weitere Mechanismen zur Synchronisation von Applikationsprozessen, die intern auf Shared-Memory basieren oder mit diesem verwandt sind. Sie sind ebenfalls über die Programmierschnittstelle für MiRPA-X einheitlich zugreifbar und können entweder blockierend oder nichtblockierend verwendet werden:

**MUTEX** Mit diesem Mechanismus kann sichergestellt werden, dass sich nur genau ein Prozess in einem kritischen Abschnitt befindet. Er nutzt den Namensdienst von MiRPA-X zur Identifizierung. Lese- oder Schreiboperationen gelten als sicher (Daten sind konsistent), sobald sich kein Prozess innerhalb des kritischen Bereichs befindet. QNX sorgt über die Prioritätsvererbung (priority inheritance) dafür, dass auch der Zugriff durch Prozesse unterschiedlicher Prioritätsebenen sicher geregelt ist.

**CONDITION** Dieser Mechanismus, der ebenfalls den Namensdienst von MiRPA-X nutzt, ermöglicht die Realisierung von prozessübergreifenden Zustandsmaschinen. Applikationsprozesse können damit blockierend oder auch nicht-blockierend das Eintreten bestimmter System- oder Applikationszustände prüfen und darauf reagieren.

**SEMAPHORE** Die Ausführung einer Applikationsfunktionalität in einem Prozess kann mit diesem Mechanismus so lange verzögert werden, bis eine vorher festgelegte Anzahl von Ereignissen eingetreten ist. Beispielsweise kann damit sichergestellt werden, dass eine Programmabarbeitung erst dann fortgeführt wird, wenn alle relevanten vorbereitenden Prozesse korrekt abgelaufen sind.

**PULS** Mit diesem Mechanismus ist es möglich, einen vorher blockierten Prozess mit einer frei wählbaren Priorität zum Ablauf zu bringen. Auf diese Weise lassen sich feste Steuerungszyklen oder notwendige Sicherheitsfunktionen deterministisch realisieren.

Bei der Realisierung von harter Echtzeitfähigkeit innerhalb der Applikation wird vornehmlich Shared-Memory-Nutzung in Kombination mit nicht-blockierenden Synchronisationsmechanismen verwendet. Für die Sicherstellung *externer* Echtzeitfähigkeit (Kommunikation mit externen Komponenten des Steuerungssystems) kommt zusätzlich noch der Einsatz des IAP (s. Abschnitt 4.3) zum Tragen.

### Echtzeitfähige Prozessablaufsteuerung

In der derzeitig verwendeten Betriebssystemversion von QNX (Neutrino, Version 6.2) werden die Schedulingmechanismen RoundRobin und FIFO (s. Abschnitt 2.1.2) in Kombination mit 64 frei wählbaren Prioritätsebenen angeboten, um die Applikations- und Systemprozesse auf einem lokalen Rechner in ihrem Ablauf automatisch zu steuern. Um nun diese allgemeinen Schedulingmöglichkeiten zu verknüpfen, zu vereinheitlichen und dabei sicherzustellen, dass alle zum Steuerungssystem gehörenden Funktionalitäten modular, in sinnvoller Weise und in garantierter Echtzeit kooperieren können, ergänzt MiRPA-X den QNX-Scheduler um eine tokenbasierte Ablaufsteuerung. Neben Funktionen zum Organisieren von Prozessabläufen und zum dynamischen Modifizieren dieser Prozessabläufe übernimmt er notwendige Sicherheitsfunktionen, wie z.B. Benachrichtigungen, falls ein Echtzeitprozess nicht innerhalb eines Applikationszyklus erfolgreich beendet werden konnte. Die dafür notwendigen Funktionalitäten sind über die bereitgestellte API konfigurierbar. Das Systemmodell für den Scheduler von MiRPA-X ist als eine Kombination von zeit- und ereignisgesteuertem QNX Scheduling realisiert (s. Abschnitt 2.1.3).

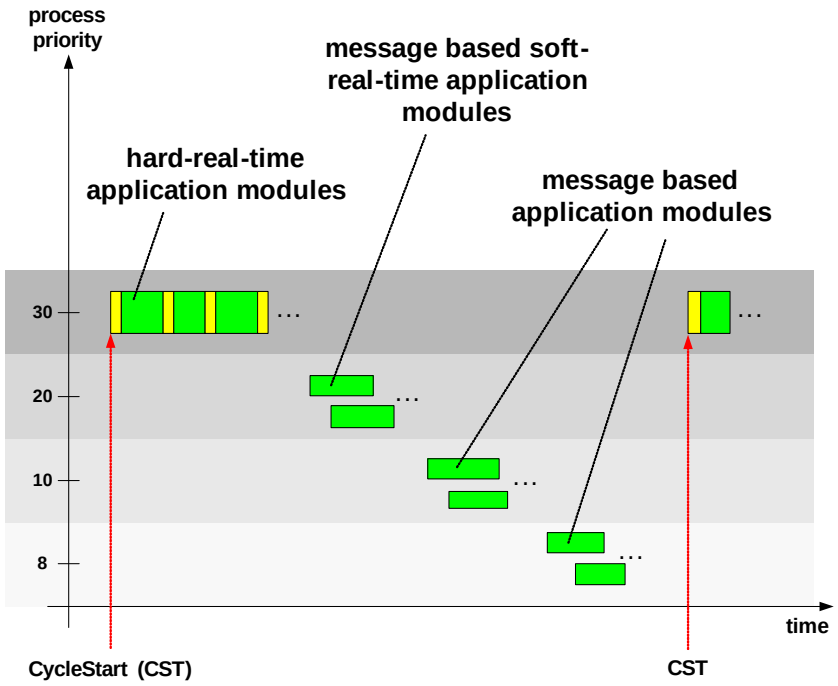



Abbildung 4.4: Ausführungsebenen des Schedulers von MiRPA-X

Das Scheduling, welches auf diese Weise speziell für zyklusbasierte Roboter-Steuerungsanwendungen geeignet ist, setzt sich damit aus zwei Komponenten zusammen: klassisches FCFS und RR für weiche Echtzeit auf unterschiedlichen Prioritätsebenen einerseits und tokenbasiertes statisches Scheduling auf hoher Prioritätsebene für harte Echtzeit andererseits. Während sich der statische Zeitbezug für die hochprioren und hardwarenahen Steuerungsebenen eignet, wird der dynamische Teil für die Realisierung von Funktionalitäten auf mittlerer (nachrichtenbasierter) Steuerungsebene verwendet.

Abbildung 4.4 stellt ein zeitlich und in Prioritätsstufen unterteiltes Ablaufdiagramm für die in Abbildung 4.2 dargestellten Applikationsmodule dar. Dieser geordnete Programmablauf wird über den *Scheduler* von MiRPA-X ermöglicht, der der laufenden Applikation ein zyklisches Zeitraster überlagert und somit das Erfüllen von harten Echtzeitbedingungen ermöglicht. Die Ablaufreihenfolge der Funktionalitäten kann dabei dynamisch vom Anwender modifiziert werden.

Der Scheduler von MiRPA-X ist frei mit einem beliebigen externen oder internen Taktgeber verknüpfbar und leitet aus diesem einkommenden Takt Startimpulse für Applikationsfunktionalitäten ab. Je nach Konfiguration und Priorisierung über den Anwender wird damit im laufenden Betrieb eine Ablaufsteuerung realisiert, die in einem deterministischen Echtzeitkontext stattfindet und von den in ihrer Priorität untergeordneten nachrichtenbasierten Vorgängen nicht unterbrochen werden kann.

Wenn MiRPA-X auf dem Zielsystem gestartet wird, beginnt zunächst die Konfigurationsphase für den Scheduler. Dabei werden die zur Ausführung erwünschten Prozessidentifikation und -reihenfolge aus einer Konfigurationsdatei herausgelesen. Diese Pro-

zesskonfiguration kann jederzeit in der Konfigurationsdatei geändert werden und wird bei einem Neustart des Schedulers aktiviert. Außerdem besteht die Möglichkeit, die Konfigurationsänderungen des Schedulers auch zur Laufzeit desselben durchzuführen, um damit die Reaktion auf Applikationsdaten zu ermöglichen (z.B. Austausch oder Verschiebung eines Echtzeit-Reglerprozesses, wenn ein neuer Sensor erkannt wird). Sind alle vorgesehenen Echtzeitprozesse des Schedulers im System vorhanden (Start erfolgt über MiRPA-X oder hündisch), so warten diese fortan auf das Eintreffen des Tokens, der nach dem Startkommando vom Scheduler an die ausgewählten Prozesse versendet wird.

Zur Laufzeit des Schedulers werden die Prozesse der harten Echtzeit sequentiell, gemäß der Konfiguration, auf hoher Prioritätsebene abgearbeitet. Dabei wird der jeweilige Zyklusstart von einem externen Synchronisierungsimpuls (hier von dem Kommunikationssystem) vorgegeben. Unabhängig von nachrichtenbasierten Vorgängen auf unteren Prioritätsebenen wird der Token nun zwischen den Prozessen weitergegeben, bis alle Echtzeitprozesse des aktuellen Zyklus abgearbeitet sind. Währenddessen tauschen diese Prozesse Daten über Shared-Memory-Bereiche aus oder stimmen ihre Algorithmen blockierungsfrei auf Zustandsvariablen (CONDITION) ab. Bei der Verwendung rotierender Berechtigungen der Prozesse werden nur dann Datenintegrität sichergestellt und Konkurrenzsituationen vermieden, wenn der jeweils gelesene und geschriebene Speicherbereich ausschließlich von Echtzeitprozessen zugegriffen wird. Im Fall, dass auch unterbrechbare Applikationsprozesse unterer Prioritätsschichten auf diese Speicherbereiche zugreifen, muss von Anwenderseite her sichergestellt werden, dass die notwendigen Synchronisationsmechanismen (MUTEX, CONDITION) von MiRPA-X verwendet werden. Das Betriebssystem QNX unterstützt diese Mechanismen durch die sogenannte Prioritätsvererbung (s. Abschnitt 2.2.4). Sobald alle Echtzeitprozesse ihre Arbeit für den jeweiligen Zyklus beendet haben, ist die verbleibende Zykluszeit frei, um von den nachrichtenbasierten Prozessen (vorrangig die Prozesse der weichen Echtzeit) genutzt zu werden. Sobald ein neuer Zyklusstart signalisiert wird, startet die Abarbeitung der sequentiellen Echtzeitprozesse erneut.

#### Error-Handling

Der fehlerfreie Applikationsablauf eines Steuerungssystems kann niemals vollständig garantiert werden. Programmierfehler innerhalb der Applikationsmodule, ein Abriss der externen Kommunikation und unsachgemäße Nutzung angebotener Kommunikationsmechanismen sind nur einige wahrscheinliche Fehlermöglichkeiten. Für eine Sicherstellung und Überprüfung von Dateninhalten ist MiRPA-X nicht vorgesehen. Allerdings werden einige Reaktionsmechanismen angeboten, die das Erkennen von fehlerhaften Abläufen innerhalb der Kommunikation an entsprechende Applikationsmodule weiterleiten können.

So wird durch die Implementierung eines Watchdog-Timers für jede REQUEST-Anfrage vermieden, dass der anfragende Client länger als beabsichtigt auf die Beantwortung warten muss, sollte der entsprechende Server nicht antworten. Bei Überschreiten der spezifizierten maximalen Wartezeit erfolgt eine entblockierende Nachricht an den Client, so dass alternative Maßnahmen ergriffen werden können. Der

ObjectServer bleibt allerdings trotz allem empfangsbereit für andere Anfragen.

Wenn ein Serverprozess, beispielsweise durch einen Applikationsfehler, unbeabsichtigt beendet wird, wird dieses automatisch im ObjectServer registriert und alle weiteren Anfragen für den beendeten Serverprozess dem jeweiligen Clientprozess gegenüber negativ beantwortet. Dadurch werden zeitraubende Testaktivitäten innerhalb der ObjectServers vermieden.

Das Überschreiten der Zykluszeit durch einen Echtzeitprozess des Schedulers ist auf eine Fehlfunktion innerhalb des Echtzeitprozesses zurückzuführen. Dieses Ereignis triggert automatisch eine „Notfallfunktion“, die auf Applikationsebene zu realisieren ist.

### 4.2.2 Realisierung MiRPA-X

In diesem Abschnitt wird auf die programmiertechnische Realisierung von MiRPA-X eingegangen. Je mehr Nutzerprozesse an der zentralen Verwaltung über eine Middleware teilnehmen sollen, um so größer ist auch die zu verwaltende Datenmenge und tendenziell auch der zur Verwaltung benötigte Zeitbedarf für einzelne Funktionalitäten der Middleware selbst. Eine programmiertechnische Realisierung stellt damit immer einen Kompromiss zwischen notwendiger Ablaufgeschwindigkeit und benötigtem Arbeitsspeicher dar. Der Schwerpunkt im Bereich der Verwaltung von Echtzeitprozessen und -ressourcen liegt so auf der Verwendung von zur Laufzeit statischen Listen und Hashwerten. Sie werden während der Konfigurationsphase festgelegt und bedingen einen konstanten Zeitaufwand unabhängig von der tatsächlichen Anzahl registrierter Applikationsprozesse.

Mit den hier dargestellten Diagrammen soll eine generelle Arbeitsweise der entwickelten Middleware MiRPA-X verdeutlicht werden, während zur ausführlichen Dokumentation ihrer Nutzung auf die 80-seitige API-Dokumentation verwiesen wird.

Abbildung 4.5 zeigt die bei der Anwendung von MiRPA-X verwendete Prozess- und Threadstruktur. Der MiRPA-X-Serviceprozess nimmt Aufträge von umliegenden Prozessen in Form von Nachrichten an und realisiert seine Verwaltungs- und Kommunikationsaufgaben über den *OBJECTSERVER*-Thread. Dieser nimmt temporär die Priorität des anfragenden Prozesses an und leitet die Anfrage entweder an einen weiteren Prozess oder führt die Anfrage selbst aus. Zur Ausführung an den ObjectServer selbst gerichteter Nachrichten wird ein *CONTROL*-Thread verwendet, der die Anfragen zunächst in einer Liste speichert und dann nacheinander auf niedriger Prioritätsebene ausführt. Über einen *MENU*-Thread können Aufträge auch direkt vom Benutzer/Programmierer an den ObjectServer übermittelt werden. Ein Auswahlfenster begrenzt die Eingabemöglichkeiten dabei auf sinnvolle Größen.

Im jeweiligen Benutzerprozess (Server) werden alle vom ObjectServer weitergeleiteten Anfragen über den *SERVER-RECEIVE*-Thread empfangen. Dieser Thread gehört zur API der Middleware und leitet die Anfrage erst dann in den vom Benutzerprozess zugreifbaren Arbeitsspeicher, wenn der Empfang der Anfrage an den ObjectServer rückbestätigt worden ist. Auf diese Weise wird sichergestellt, dass der ObjectServer

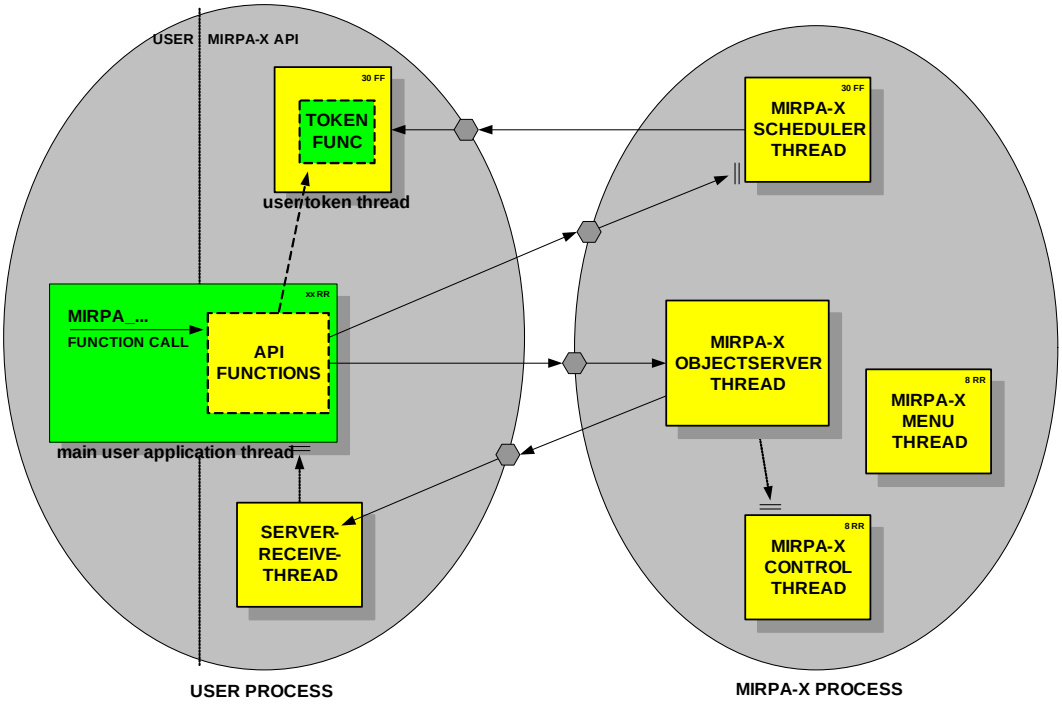



Abbildung 4.5: Struktur der Arbeits-Threads von MiRPA-X

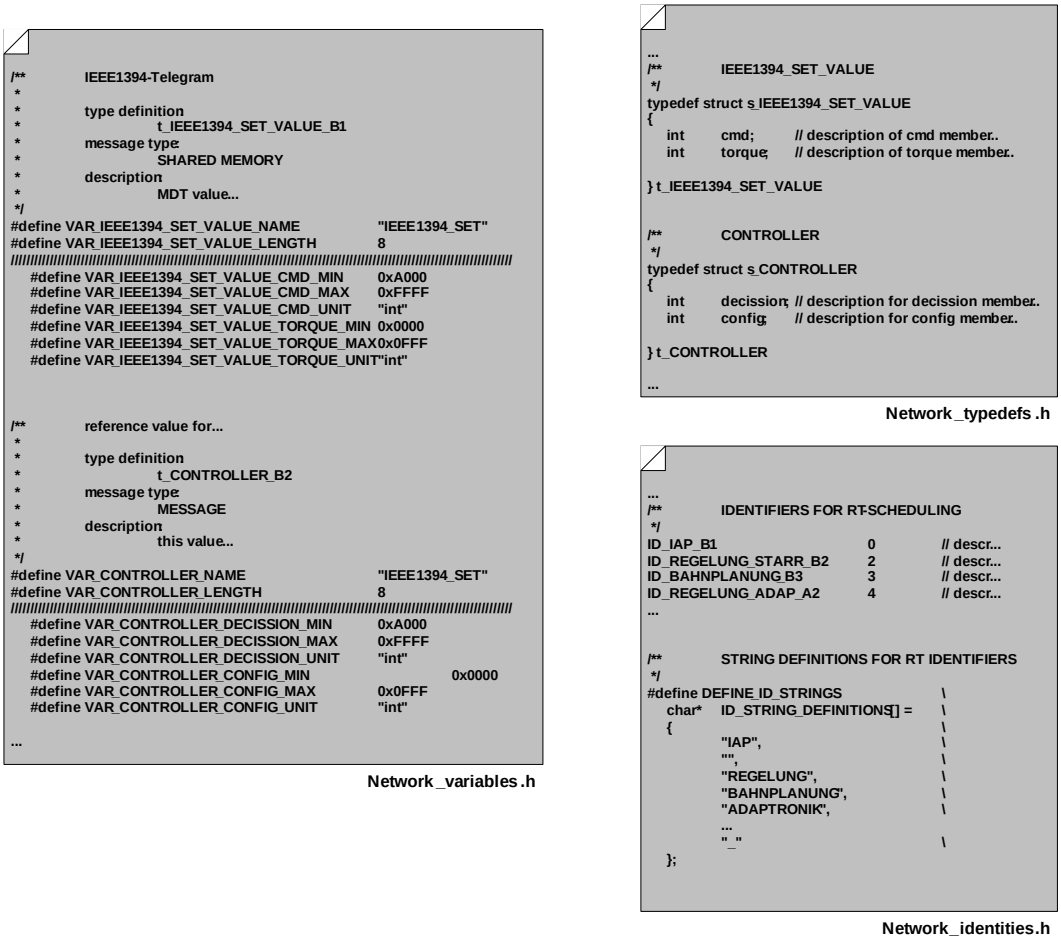
nicht durch eine von einem User-Thread verursachte Fehlfunktion blockiert werden kann. Der *USER*-Thread des Benutzers kann dann auf die Anfrage reagieren (Server); dabei bleibt seine Priorität unverändert. Im Fall eines Clients werden Anfragen von hieraus direkt über Funktionsaufrufe an die API initiiert und als Nachrichten an den ObjectServer-Thread übertragen, der wieder temporär die Priorität des Aufrufers übernimmt, bis die Nachricht weitergeleitet wurde.

Der MiRPA-X-*SCHEDULER*-Thread koordiniert den Echtzeit-Prozessablauf auf der höchsten Applikations-Prioritätsebene. Dabei wird zur Laufzeit des Schedulers eine jeweilige *TOKEN-FUNKTION* der Applikation innerhalb eines *TOKEN*-Thread zur Ausführung gebracht, wenn dieser über ein entsprechendes Token vom Scheduler-Thread benachrichtigt wird. Der Token-Thread selbst wird über die MiRPA-X-API von der Applikation konfiguriert.

In den folgenden Abschnitten wird im Detail auf die Umsetzung der Funktionalitäten eingegangen. Angefangen beim Start, der Konfigurierung und einer Vorstellung wesentlicher Funktionen der Benutzer- oder Programmierschnittstelle, wird im Anschluss der Objektzusammenhang dargestellt.

### Dateien zur Konfigurierung und zum Start

Die Konfigurierung von MiRPA-X geschieht in zwei Stufen. Zunächst wird *während des Kompilierens* festgelegt, welche echtzeitrelevanten Anwendungsmodule (Echtzeit-Threads) zur Maximalkonfiguration der Gesamtsteuerung gehören, welche Daten-



```
/**         IEEE1394-Telegram
 *
 *          type definition
 *                  t_IEEE1394_SET_VALUE_B1
 *          message type:
 *                  SHARED MEMORY
 *          description:
 *                  MDT value...
 */
#define VAR_IEEE1394_SET_VALUE_NAME             "IEEE1394_SET"
#define VAR_IEEE1394_SET_VALUE_LENGTH           8
////////////////////////////////////////////////////////////////////////////////////////////////
   #define VAR_IEEE1394_SET_VALUE_CMD_MIN       0xA000
   #define VAR_IEEE1394_SET_VALUE_CMD_MAX       0xFFFF
   #define VAR_IEEE1394_SET_VALUE_CMD_UNIT      "int"
   #define VAR_IEEE1394_SET_VALUE_TORQUE_MIN 0x0000
   #define VAR_IEEE1394_SET_VALUE_TORQUE_MAX0x0FFF
   #define VAR_IEEE1394_SET_VALUE_TORQUE_UNIT"int"



/**         reference value for...
 *
 *          type definition
 *                  t_CONTROLLER_B2
 *          message type:
 *                  MESSAGE
 *          description:
 *                  this value...
 */
#define VAR_CONTROLLER_NAME                     "IEEE1394_SET"
#define VAR_CONTROLLER_LENGTH                   8
////////////////////////////////////////////////////////////////////////////////////////////////
   #define VAR_CONTROLLER_DECISSION_MIN         0xA000
   #define VAR_CONTROLLER_DECISSION_MAX         0xFFFF
   #define VAR_CONTROLLER_DECISSION_UNIT        "int"
   #define VAR_CONTROLLER_CONFIG_MIN                    0x0000
   #define VAR_CONTROLLER_CONFIG_MAX            0x0FFF
   #define VAR_CONTROLLER_CONFIG_UNIT           "int"

...
```

Network_variables.h

```
...
/**         IEEE1394_SET_VALUE
 */
typedef struct s_IEEE1394_SET_VALUE
{
   int      cmd;        // description of cmd member..
   int      torque;     // description of torque member..

} t_IEEE1394_SET_VALUE


/**         CONTROLLER
 */
typedef struct s_CONTROLLER
{
   int      decission;  // description for decission member..
   int      config;     // description for config member..

} t_CONTROLLER

...
```

Network_typedefs.h

```
...
/**         IDENTIFIERS FOR RT-SCHEDULING
 */
ID_IAP_B1                   0       // descr...
ID_REGELUNG_STARR_B2        2       // descr...
ID_BAHNPLANUNG B3           3       // descr...
ID_REGELUNG_ADAP_A2         4       // descr...
...


/**         STRING DEFINITIONS FOR RT IDENTIFIERS
 */
#define DEFINE_ID_STRINGS               \
   char*    ID_STRING_DEFINITIONS[] =   \
   {                                    \
            "IAP",                      \
            "",                         \
            "REGELUNG",                 \
            "BAHNPLANUNG",              \
            "ADAPTRONIK",               \
            ...                         \
            "_"                         \
   };
```

Network_identities.h

Abbildung 4.6: Dateien zum Aufbau der statischen Systemstruktur

struktur-Definition innerhalb der Anwendung verwendet werden können und welche Variablen tatsächlich für welche Aufgabe verwendet werden. Abbildung 4.6 zeigt die drei Include-Dateien, in denen diese Informationen abgelegt werden können.

**Network_identities.h** Diese Datei wird verwendet, um die im Steuerungssystem auswählbaren Echtzeit-Threads festzulegen. Im oberen Teil der dargestellten Datei werden dazu Thread-Identifikationsnummern definiert. Diese Identifikationsnummern werden zur Laufzeit des Schedulers verwendet, um den jeweiligen Token-Thread zu adressieren. Im unteren Teil der Datei wird festgelegt, welcher Threadbezeichnung die jeweilige Identifikationsnummer zugeordnet ist.

**Network_typedefs.h** In dieser Datei werden die Datenstrukturen spezifiziert, wie sie innerhalb der Steuerungsanwendung für die Nutzung von Nachrichten verwendet werden sollen. Eine solche Festlegung dient der Einheitlichkeit und Austauschbarkeit von Daten auf dem lokalen Steuerungsrechner sowie zwischen unterschiedlichen, auch hardwarespezifischen, Recheneinheiten.

**Network_variables.h** Diese Datei beinhaltet die Namen der tatsächlichen im Steuerungssystem über MiRPA-X vermittelten Nachrichten, ihrer Wertebereiche und

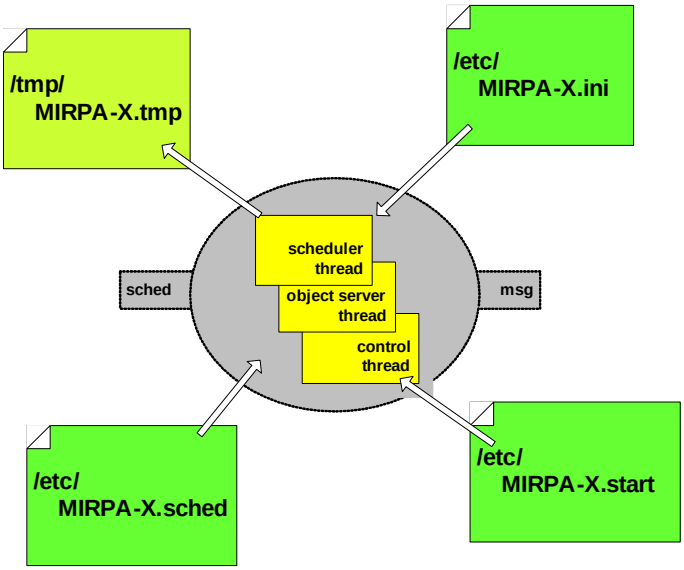



Abbildung 4.7: Dateien für den dynamischen Start von MiRPA-X

physikalischen Einheiten. Durch diese Aufstellung ist jedem Prozess im Steuerungssystem jede relevante Nachricht zugänglich. Es lassen sich so dynamisch auch skalierte Visualisierungen für Dateninhalte realisieren.

Neben dieser statischen Konfigurierung zur Festlegung des Maximalausbaus des Steuerungssystems wird auch eine dynamische Konfigurierungsmöglichkeit zur Verfügung gestellt, die *erstmalig beim Start* von MiRPA-X eine Einstellung der Laufzeiteigenschaften ermöglicht. Abbildung 4.7 stellt die für diese Art der Konfiguration verwendeten Dateien dar. Es handelt sich ausschließlich um ASCII-Dateien.

**/etc/MIRPA-X.ini** In dieser Datei werden globale Einstellungen vereinbart, die den Auffindungsort benötigter Dateien, den Umfang anzulegender statischer Strukturen und das ObjectServer-Verhalten bei kollidierenden Anfragen betreffen. Diese Datei wird jeweils beim Start von MiRPA-X einmalig eingelesen und ausgewertet.

**/etc/MIRPA-X.sched** Über diese Datei kann erstmals festgelegt werden, in welcher Reihenfolge die Echtzeit-Funktionalitäten (Token-Threads) vom Scheduler während der Echtzeitphase eines Kommunikationszyklus aufgerufen werden. Dieser Ablauf kann dynamisch über die API verändert werden.

**/etc/MIRPA-X.start** Mit Hilfe dieser Datei können die zum Steuerungssystem gehörenden Prozesse automatisch gestartet werden. Sie verbinden sich anschließend automatisch mit dem ObjectServer, melden (im Fall eines Servers) Nachrichten an und reservieren Shared-Memory-Bereiche.

**/tmp/MIRPA-X.tmp** Hier werden während der Initialisierung von MiRPA-X Kanal-Identifikationen gespeichert, die während des Anmeldevorgangs anderer Prozesse von deren API benötigt werden.

| | |
|---|---|
| **MIRPA_SendControlMsg()** | Versenden einer CONTROL-Nachricht an den ObjectServer, um Informationen über bereits registrierte Nachrichten , Mutexe, Shared-Memory,... einzuholen |
| **MIRPA_AccessSharedMem()** | Zugriff auf einen bestimmten Shared-Memory-Bereich beantragen und Zeiger darauf von MiRPA-X erhalten |
| **MIRPA_SendCommandMsg()** | Absenden einer COMMAND-Nachricht, um Aktivitäten in einem oder mehreren Servern auszulösen |
| **MIRPA_SendRequestMsg()** | Absenden einer REQUEST-Nachricht, um Aktivitäten in einem Server auszulösen, die über eine ANSWER-Nachricht unter gleichem Namen beantwortet werden müssen |
| **MIRPA_ReceiveMsg()** | Blockierend auf das Eintreffen einer Nachricht warten (Server) |
| **MIRPA_ReplyMsg()** | Antworten auf eine empfangene REQUEST-Nachricht (Server) |
| **MIRPA_WaitForCondition()** | Blockieren, so lange eine bestimmte Bedingung (CONDITION) noch nicht erfüllt ist |
| **MIRPA_SignalNewCondition()** | Eine bestimmte Bedingung (CONDITION) setzen und damit alle darauf wartenden Prozesse benachrichtigen |
| **MIRPA_EnterMutex()** | Einen kritischen Bereich betreten, um Ressourcen sicher nutzen zu können |
| **MIRPA_LeaveMutex()** | Den zuvor betretenen kritischen Bereich verlassen, um die Ressource anderen Prozessen zugänglich zu machen |

Abbildung 4.8: Funktionen der Benutzerschnittstelle (1)

### Programmierschnittstelle, API

MiRPA-X verwendet verschiedene Nachrichtentypen, um funktionsspezifische Kommunikation zwischen Prozessen zu realisieren. Diese Nachrichtentypen sind weiter oben in diesem Abschnitt beschrieben. Als Grundlage für eine Programmier- oder Benutzerschnittstelle (API) eignen sie sich allerdings nur bedingt, da dies für den Benutzer die genaue Kenntnis der Funktionsweise der Middleware voraussetzen würde. Darum wurde die API nicht nachrichtenbasiert oder funktionsbezogen realisiert, sondern verwendet aufgabenbezogene Funktionsbezeichnungen und einheitliche Über- und Rückgabeparameter. Sie kann damit nahezu intuitiv genutzt werden. Als Übergabeparameter verwendet die API ausschließlich Referenznummern (Handles) und Speicheradressen, so dass Nutzerdaten über Listenfunktionalitäten adressiert und Daten während der Konfigurationsphase organisiert werden können. Zyklische und zeitraubende Kopiervorgänge von Anwendungsdaten zwischen etwaigen Softwareschichten entfallen und Applikationsvorgänge lassen sich über programmierbare Identifikationsnummern automatisieren. Als direkte Rückgabeparameter werden in der Regel Fehlercodes verwendet, um Ablaufsteuerungen in Abhängigkeit vom Funktionserfolg realisieren zu können.

Abbildung 4.8 zeigt wesentliche Funktionen, die während der Laufzeit des Roboters zur Nachrichtenübertragung, Shared-Memory-Nutzung und Prozesssynchronisation verwendet werden. Sie sind innerhalb einer C-Bibliothek realisiert und ermöglichen die Nutzung auch innerhalb einer C++ Umgebung, wie es innerhalb des SFB 562 gelegentlich üblich war. Die hier dargestellten Funktionen sind Teil einer 40 Funktionen umfassenden API. Dort sind weitere Funktionen zur Konfigurierung vorhanden. Alle

| | |
|---|---|
| **MIRPA_StartToken()** | Starten des MiRPA-X-Schedulers. Hiermit wird der Token an den ersten wartenden Echtzeitprozess weitergegeben. |
| **MIRPA_ScheduleTokenThread()** | Einfügen eines neuen TokenThreads (Echtzeitprozess) in die Liste der abzuarbeitenden Prozesse des Schedulers |
| **MIRPA_RemoveTokenThread()** | Entfernen eines TokenThreads aus der Liste der abzuarbeitenden Prozesse des Schedulers |
| **MIRPA_ReplaceTokenThread()** | Ersetzen eines TokenThreads durch einen anderen. Hiermit können sehr dynamische, applikationssbedingte Scheduling-strategien umgesetzt werden |
| **MIRPA_StopToken()** | Festhalten des Tokens und damit Beenden des Schedulers |
| **MIRPA_InitTokenThread()** | Registrieren des aufrufenden Prozesses beim Scheduler |
| **MIRPA_WaitForToken()** | Blockieren, bis der Token empfangen wird |
| **MIRPA_ReleaseToken()** | Token an den nächsten Echtzeitprozess weitergeben |
| **MIRPA_ExitToken()** | Aufrufender Prozess verlässt die Scheduler-Abarbeitung |

Abbildung 4.9: Funktionen der Benutzerschnittstelle (2)

Funktionen sind in einer ausführlichen API-Dokumentation beschrieben.

Abbildung 4.9 zeigt einen Ausschnitt aus der API des Schedulers, der in seiner momentanen Implementierung insgesamt 12 MiRPA-X-Funktionen umfasst. Bei der Verwendung von MiRPA-X-Funktionen innerhalb der Token-Threads (Anwendungsfunktionalitäten der harten Echtzeitebene) ist zu beachten, dass es hier Beschränkungen gibt, die vor allem die Nutzung des blockierenden Nachrichtentransfers und weiterer blockierender Synchronisationsmechanismen betreffen. Die API-Dokumentation enthält daher eindeutige Hinweise zu diesen Nutzungsbeschränkungen.

### Intern verwendete Datenstrukturen

Abbildung 4.10 zeigt die zur Verwaltung von Prozessen und Kommunikationsobjekten verwendeten internen Datenstrukturen des ObjectServers von MiRPA-X. Für jeden jemals mit dem ObjectServer in Verbindung getretenen Prozess existiert ein entsprechender ProcessInfoBlock `t_UserProcessInfo`, der sowohl die eindeutige Zuordnung von Nachrichten als auch die Gewinnung statistischer Informationen über Kommunikationsvorgänge zulässt. Jeder ProcessInfoBlock wird von anderen Datenstrukturen, z.B. `t_ShMemEntry` oder `t_MsgHashTab`, über einen eindeutigen Index referenziert. Auf diese Weise entsteht innerhalb des ObjectServers ein genaues Abbild des momentanen Kommunikationsnetzes und laufender Kommunikationsvorgänge.

Die aktiven Nachrichtenobjekte werden über die Datenstruktur `t_message` verwaltet. Hierin wird neben den Dateninhalten auch deren Herkunft gespeichert, so dass die Adressierung der Antwort eindeutig erfolgen kann. Die Shared-Memory-Bereiche `t_ShMemEntry` werden vom ObjectServer verwaltet. Sie dienen neben der direkten Verwendung zum generellen Datenaustausch zwischen Applikationsprozessen auch als Grundlage für darauf aufbauende Funktionen; über sie wer-

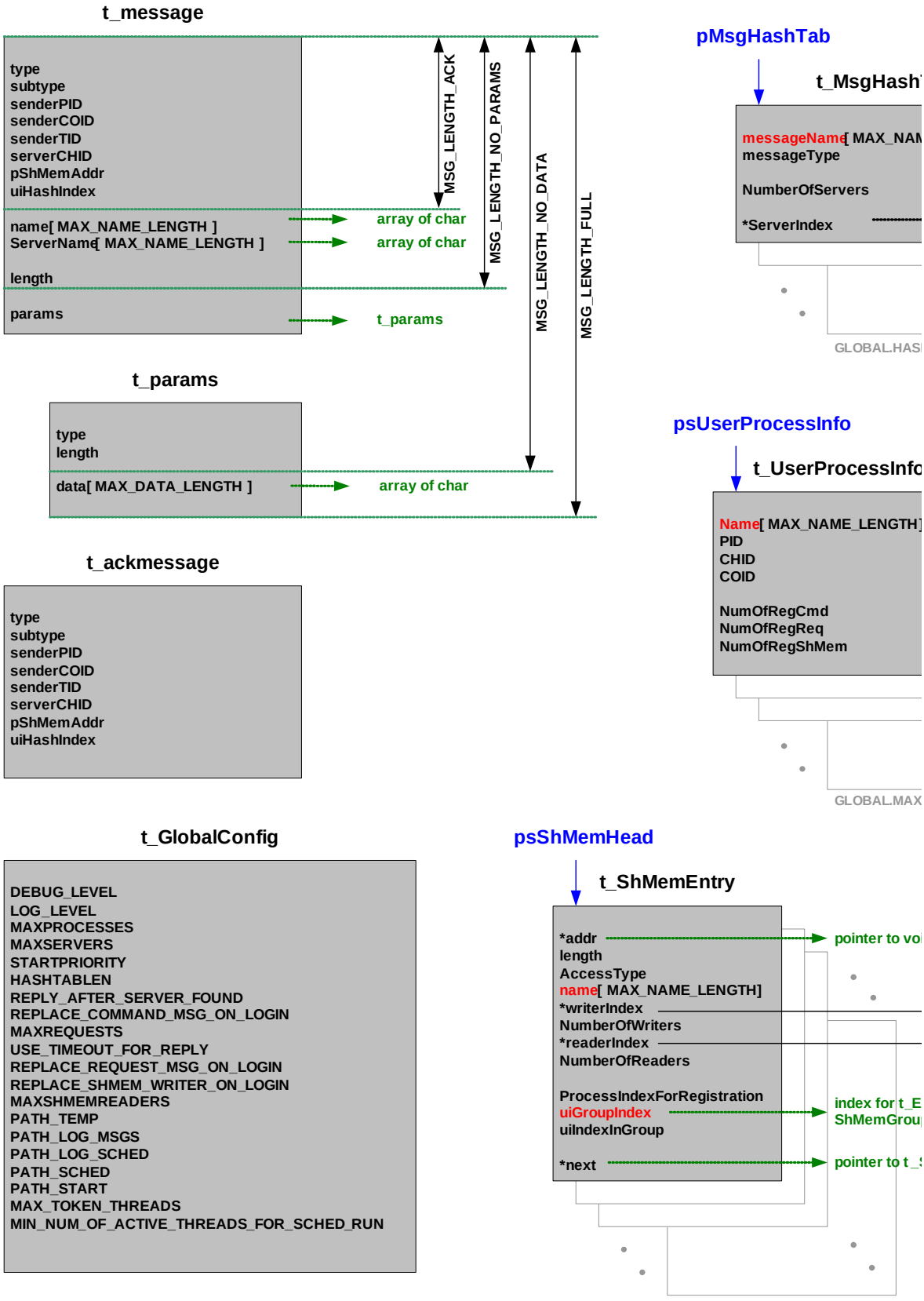
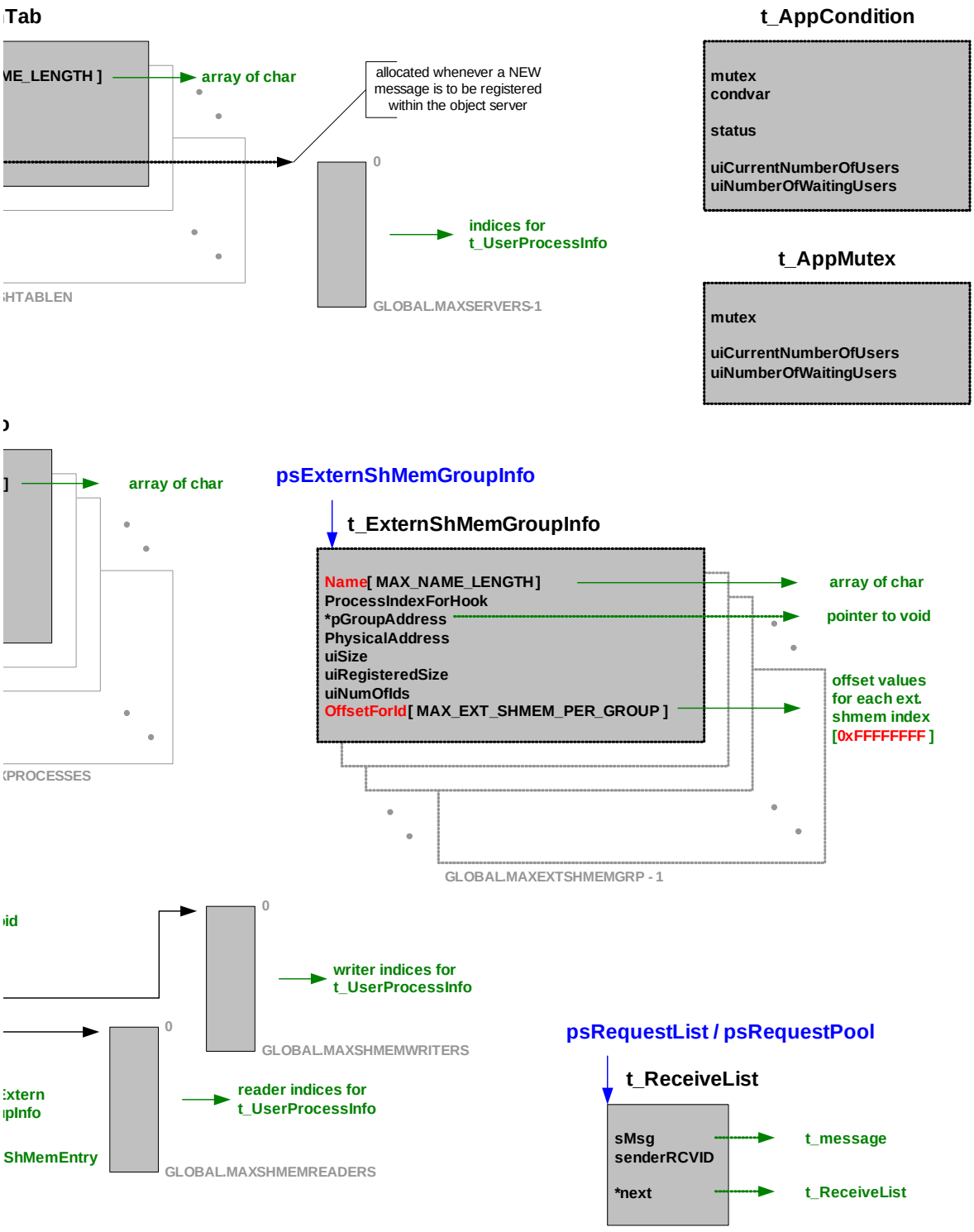



Abbildung 4.10: Datenstrukturen zur Verwaltung des ObjectServers


Tab
ME_LENGTH ]
array of char
allocated whenever a NEW message is to be registered within the object server
0
indices for t_UserProcessInfo
HTABLEN
GLOBAL.MAXSERVERS-1
t_AppCondition
mutex
condvar
status
uiCurrentNumberOfUsers
uiNumberOfWaitingUsers
t_AppMutex
mutex
uiCurrentNumberOfUsers
uiNumberOfWaitingUsers
]
array of char
PROCESSES
psExternShMemGroupInfo
t_ExternShMemGroupInfo
Name[ MAX_NAME_LENGTH ]
ProcessIndexForHook
*pGroupAddress
PhysicalAddress
uiSize
uiRegisteredSize
uiNumOfIds
OffsetForId[ MAX_EXT_SHMEM_PER_GROUP ]
array of char
pointer to void
offset values for each ext. shmem index [0xFFFFFFFF ]
GLOBAL.MAXEXTSHMEMGRP - 1
id
0
writer indices for t_UserProcessInfo
GLOBAL.MAXSHMEMWRITERS
0
xtern
pInfo
reader indices for t_UserProcessInfo
ShMemEntry
GLOBAL.MAXSHMEMREADERS
psRequestList / psRequestPool
t_ReceiveList
sMsg
senderRCVID
*next
t_message
t_ReceiveList

den auch Synchronisierungsfunktionalitäten für Mutex `t_AppMutex` und Condition `t_AppCondition` realisiert, die dann über die Variable *AccessType* festgehalten werden. Weiter können die Shared-Memory-Bereiche externen Shared-Memory-Gruppen `t_ExternShMemGroupInfo` zugeordnet werden. Über diese Gruppen lassen sich Speicherzugriffe von Hardwarekomponenten im Echtzeitkontext wesentlich beschleunigen.

Von Clients einkommende Anfragen werden in einer Liste `t_ReceiveList` als offene Nachrichten zwischengespeichert, bis eine Antwortnachricht von dem entsprechenden Serverprozess empfangen wird. Nachrichten unterschiedlicher Applikationsprozesse können sich auf diese Weise auch kreuzen, ohne dass die Zuordnung beeinträchtigt werden kann.

Zwischen Applikationsprozessen erfolgen Synchronisierungen (beispielsweise für Mutex und Condition) intern über Shared-Memory-Bereiche. Diese Bereiche werden ebenfalls über Shared-Memory-Bereiche verwaltet, in denen die Strukturen `t_AppCondition` und `t_AppMutex` definiert sind. Sie beschreiben die Synchronisierungsfunktionalität sowie auch statistische Informationen über die momentane Nutzung und können, da sie selbst im Shared-Memory liegen, auch direkt von Funktionalitäten innerhalb der Benutzerschnittstellen der Applikationen gelesen und modifiziert werden.

Die in der Datei MIRPA-X.ini definierten Einstellungen werden beim Programmstart in die Struktur `t_GlobalConfig` geladen, um danach in der gesamten MiRPA-X-Anwendung zugreifbar zu sein. Die Inhalte können jederzeit über API-Funktionen modifiziert werden, so dass eine Verhaltensänderung der Middleware während der Laufzeit möglich ist.

Abbildung 4.11 zeigt die zur Verwaltung der lokalen Kommunikationsvorgänge innerhalb der Clients und Server verwendeten Datenstrukturen der Benutzerschnittstelle. Die Datenstruktur `t_message` ist hier nicht noch einmal aufgeführt, da sie weiter oben schon vorgestellt wurde. Innerhalb der Struktur `t_CommConfig` werden die Einstellungen der lokalen Applikation zur Verwaltung von Kommunikationsvorgängen abgelegt. Neben globalen Identifikationsnummern finden sich hier der momentane Initialisierungszustand für Kommunikationsfunktionalitäten sowie Verweise auf Listen zur Verwaltung dieser Funktionalitäten. Die Verweise beziehen sich beispielsweise auf Elemente der Struktur `t_ReceiveList`, die als Nachrichten-Warteschlange realisiert ist. In diese gelangen alle vom ObjectServer an den lokalen Prozess adressierte Nachrichten.

Über die Struktur `t_MsgList` werden innerhalb der Benutzerschnittstelle alle Nachrichten verwaltet, die aktiv vom Benutzer innerhalb seiner Anwendung versendet oder empfangen werden können. Es handelt sich um vorkonfigurierte Nachrichten, die über eine Nachrichten-ID referenziert werden. Um die Arbeitsgeschwindigkeit der Benutzerschnittstelle beim Nachrichtentransfer zu maximieren, werden die Nachrichten, die vom Betriebssystem zwischen den Prozessbereichen kopiert werden, nicht weiter kopiert. Ein Zugriff auf die im lokalen Puffer gespeicherten Nachrichtendaten erfolgt dann direkt über Zeigeroperationen. Diese Zeiger werden ebenfalls in `t_MsgList` in Abhängigkeit von der jeweiligen Nachrichten-ID abgelegt.

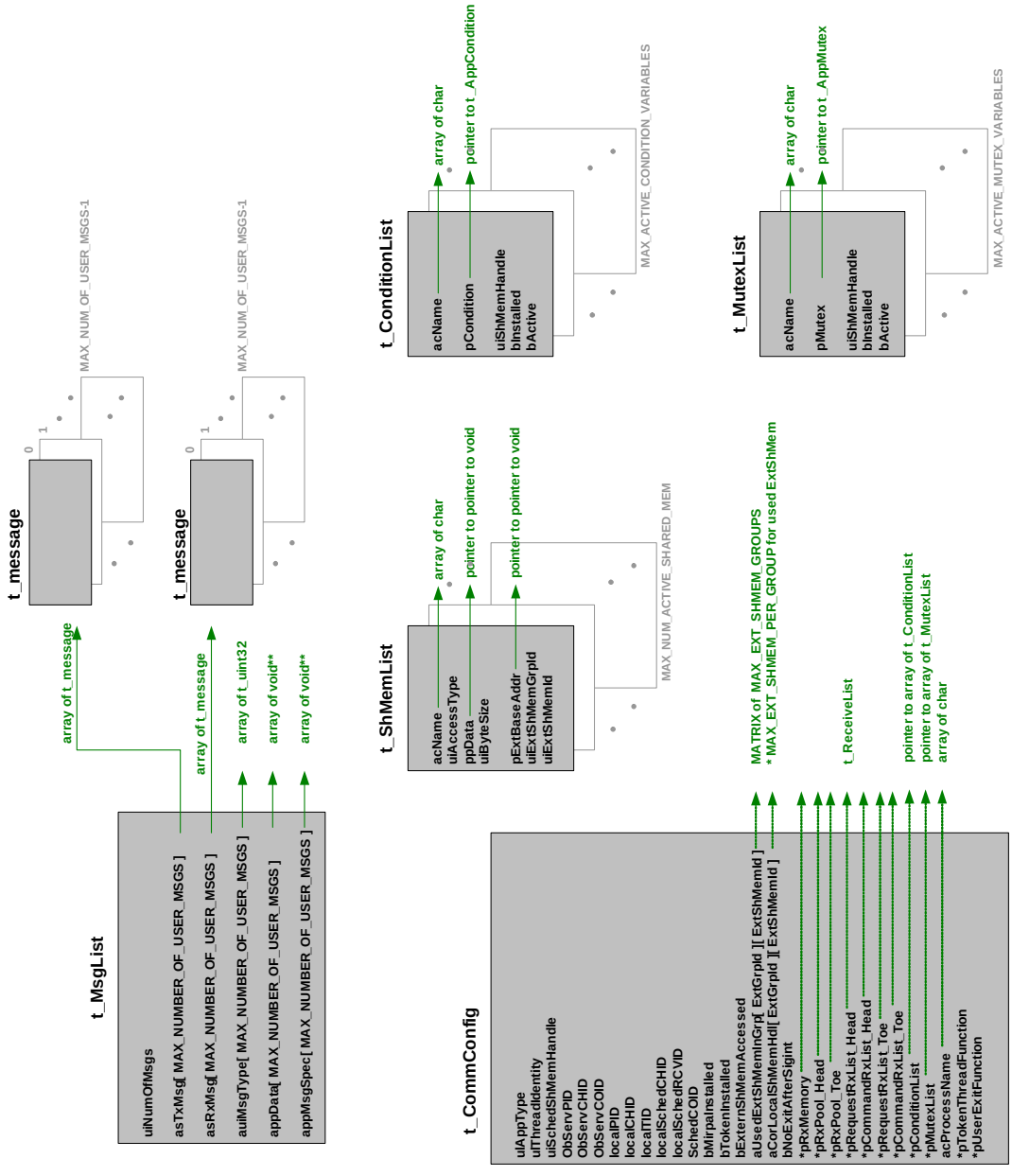



Abbildung 4.11: Datenstrukturen zur Verwaltung der Benutzerschnittstelle

Die Struktur `t_ShMemList` wird genutzt, um die vom lokalen Applikationsprozess verwendeten Shared-Memory-Bereiche zu verwalten. Dabei wird in dieser Struktur ein Zeiger auf einen Zeiger gespeichert, der sich auf den global nutzbaren, über einen einheitlichen Namen identifizierten Speicherbereich bezieht. Über Strukturen `t_MutexList` und `t_ConditionList` wird der Zugriff auf die in der gesamten Steuerungsapplikation bekannten Synchronisierungsfunktionalitäten gewährt, die ebenfalls über einen Namensdienst verwaltet werden.

In den folgenden Abschnitten erfolgt nun die detaillierte Darstellung der auf der Verwendung der vorgestellten Datenstrukturen basierenden Funktionalitäten der Middleware. Dabei soll eine ganzheitliche Darstellung dadurch erzielt werden, dass zum einen die Vorgänge innerhalb der API und zum anderen diejenigen innerhalb des ObjectServers bzw. entsprechender MiRPA-X-Threads erläutert werden. Dabei wird bewusst auf eine abstrakte Klassendarstellung verzichtet und der Fokus auf Verständlichkeit gelegt. Anhand dieser Darstellung soll die Geradlinigkeit, Leistungsfähigkeit und Stabilität des gewählten Ansatzes herausgestellt werden.

### Initialisierungsablauf

Die Middleware MiRPA-X wird über das Starten des ObjectServers im Arbeitsspeicher des lokalen Rechners installiert und läuft dann im Hintergrund ab, so lange die Steuerungsapplikation die Kommunikations- und Synchronisationsfunktionalitäten benötigt. Abbildung 4.12 zeigt das Sequenzdiagramm des ObjectServers beim Start. Zunächst werden nach der Sicherstellung, dass momentan keine weitere Middlewareinstanz aktiv ist, die Kommandozeilenparameter ausgewertet und entsprechende Bedingungsvariablen gesetzt. Zur Übernahme der vom Nutzer gewünschten globalen Einstellungen werden die Initialisierungsdatei ausgelesen und entsprechende Parameter gesetzt. Nun werden die Tabellen, Listen und Datenpuffer initialisiert, die während der Anwendung innerhalb der Steuerung mit Daten gefüllt werden sollen. Damit Applikationsprozesse später auf den ObjectServer zugreifen können, werden nun die Prozessidentifikation und Kommunikations-Kanalnummer in einer temporären Datei abgelegt.

In den folgenden Schritten werden die drei weiteren Threads der Middleware gestartet. Der im ersten Schritt auf niedriger Prioritätsebene erzeugte Control-Thread führt zunächst die Anweisungen der Start-Datei aus, in der Anwender beliebige Kommandozeilenanweisungen für den Start ablegen können. Anschließend wartet dieser Thread auf die interne Signalisierung einer neuen CONTROL-Nachricht. In einem zweiten Schritt wird nun der Scheduler-Thread auf hoher Prioritätsebene erzeugt. Dieser registriert zunächst die von ihm benötigten Speicherbereiche beim ObjectServer und initialisiert darin zu speichernde Strukturen. Für die nachfolgende Echtzeitverarbeitung erzeugt er einen eigenen Kommunikationskanal, liest und initialisiert die Start-Ablaufkonfiguration aus der Scheduler-Konfiguraions-Datei (s. Abbildung 4.6). Anschließend wartet der Thread auf ein Scheduler-Kommando, das entweder vom Anwender oder vom ObjectServer erfolgen kann. Eine detaillierte Beschreibung des Starts und der weiteren Operation des Schedulers erfolgt in diesem Abschnitt ab Seite 122. Im dritten Schritt wird nun, falls dieses in den Kommandozeilenparametern so

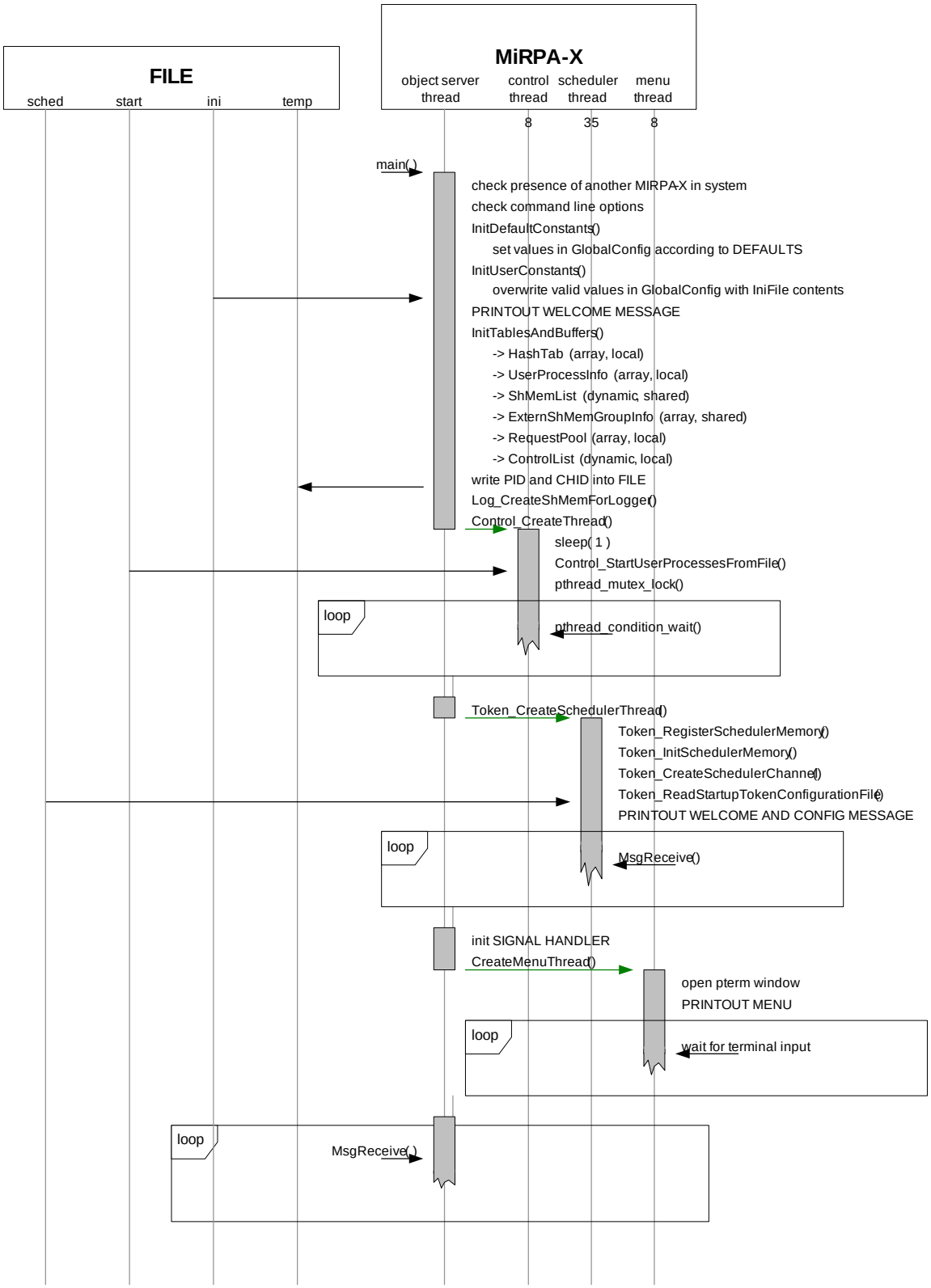



Abbildung 4.12: Ablauf beim Start des ObjectServers

gewünscht wurde, ein Menü-Thread auf niedriger Prioritätsebene gestartet. Hiermit lassen sich Nachrichten online generieren und direkt an den ObjectServer versenden.

Sobald alle Threads gestartet wurden, begibt sich der ObjectServer in einen Wartezustand für einkommende Nachrichten. Nun können Applikationsprozesse ihre Arbeit aufnehmen. Abbildung 4.13 zeigt dazu ein Sequenzdiagramm, das die Initialisierung eines jeden Clients bzw. Servers beschreibt.

Die Initialisierung wird innerhalb der Applikationsprozesse mit Aufruf der Funktion **MIRPA_Init()** durchgeführt. Zunächst erfolgt die Initialisierung benötigter Modulvariablen und die Erzeugung lokaler Strukturen. Anschließend wird die Kommunikationskonfiguration des ObjectServers aus der temporären Datei des ObjectServers ausgelesen und darüber ein gültiger Kommunikationskanal aufgebaut. Je nach Prozesstyp (lokale Applikation ist entweder Client oder Server) erfolgt die weitere Initialisierung unterschiedlich. Für Server wird nun ein separater Kommunikationskanal zum ObjectServer aufgebaut und anschließend ein Empfangs-Thread erzeugt, der über diesen Kanal auf Nachrichten warten und diese empfangen kann. Nach der Erzeugung weiterer Strukturen, die für Client und Server gleich sind, werden noch Shared-Memory-Bereiche beim ObjectServer angemeldet und die Kontrolle wieder an den Aufrufer zurück gegeben.

Für jede Funktionalität und jeden Funktionsaufruf innerhalb der Initialisierungsphase der Middleware existieren weitere detaillierte Sequenz- oder entsprechende Flussdiagramme. Eine Verknüpfung ist, wie in Abbildung 4.13 für die Funktion `LoginShMem()`, immer über das Schlagwort „ref“ angedeutet. Aus Platzgründen wird an dieser Stelle der generelle Initialisierungsprozess nicht weiter behandelt. Weiterführende Informationen sind der API-Dokumentation sowie der begleitenden Diagrammsammlung zu entnehmen.

Im Folgenden werden zum Verständnis der grundlegenden Funktionalitäten Sequenzdiagramme für Nachrichtenkonfigurierung, Nachrichtentransfer sowie Synchronisations- und Schedulerfunktionen dargestellt.

### Vorgänge zur NACHRICHTEN-Übertragung

Abbildung 4.14 zeigt das Sequenzdiagramm, über das MiRPA-X Nachrichten für einen lokalen Applikations-Client erzeugt, die später verschickt werden können. Über die Trennung von Erzeugen und Versenden einer Nachricht ist es möglich, die Zeit, die später zum Versenden der Nachricht benötigt wird, zu minimieren.

Innerhalb der API-Funktion zum Erzeugen einer Nachricht werden nach dem Test einiger Beschränkungsbedingungen der von der Applikation gewünschten Nachricht eine prozesslokale Identifikationsnummer zugeordnet und ein eindeutiger, globaler Hashindex berechnet. Nun wird unterschieden, ob eine COMMAND- oder eine REQUEST-Nachricht erzeugt werden soll. Im ersten Fall wird ein Nachrichten-Ausgangspuffer mit spezifischen Parametern für den Versand vorbereitet. Der von der Applikation bereitgestellte Datenzeiger wird auf den Datenbereich innerhalb des Nachrichten-*Ausgangs*puffers gesetzt. Dadurch berechnet die Applikation ihre Applikationsdaten

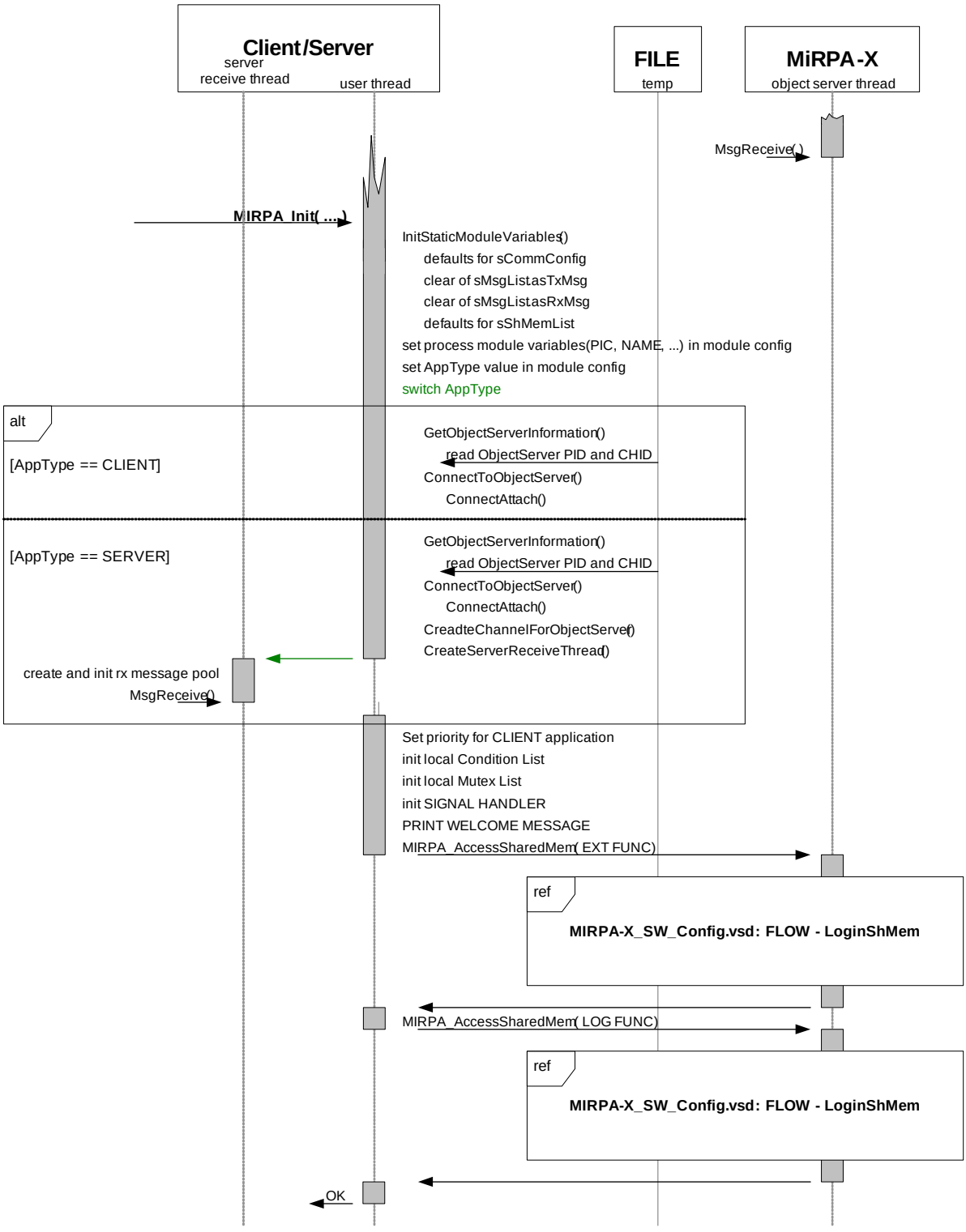



Abbildung 4.13: Ablauf beim Start des Anwendungsprozesses

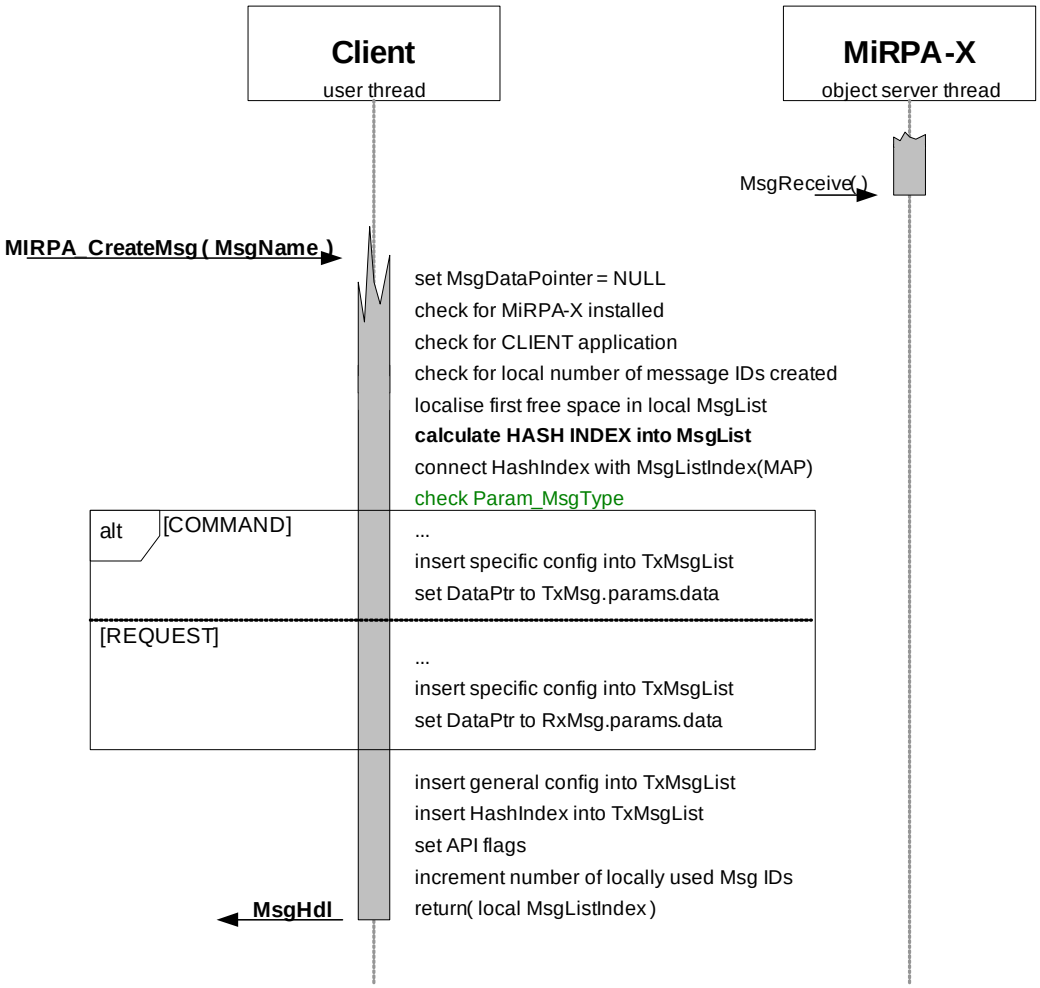



Abbildung 4.14: Vorgänge beim Erzeugen eines Nachrichtenobjekts

bereits in einem Datenbereich innerhalb der zu versendenden Nachricht, so dass bei einem späteren Versendewunsch kein zusätzlicher Kopiervorgang mehr erfolgen muss. Im Fall des Erzeugens einer REQUEST-Nachricht wird der Datenzeiger stattdessen auf den entsprechenden Speicherbereich innerhalb des Nachrichten-*Eingangs*puffers gesetzt. Dadurch erfolgt der Empfang einer ANSWER-Nachricht quasi direkt in die entsprechende Variable der Applikation hinein, ohne einen weiteren Kopiervorgang zu erfordern.

Während der Erzeugung einer Nachricht erfolgt keine Kommunikation mit dem ObjectServer. Alle zum Versenden einer Nachricht benötigten Daten werden lokal im Applikationsspeicher abgelegt und von dort auch gelesen.

Abbildung 4.15 zeigt das Sequenzdiagramm, über das bei MiRPA-X Nachrichten für einen lokalen Applikations-Server angemeldet werden. Nach erfolgreicher Registrierung wird jede entsprechende COMMAND- oder REQUEST-Nachricht, die von einem anderen Applikations-Client stammt, an einen dafür registrierten Applikations-Server weitergeleitet.

Bis zur Zuordnung von lokaler Identifikationsnummer zu berechnetem globalem Hashindex verhält sich die API-Funktion zum Registrieren einer Nachricht analog zu derjenigen zur Erzeugung einer Nachricht. Nun wird allerdings eine CONFIG-Nachricht an den ObjectServer verschickt, um den aktiven Applikations-Serverprozess dort für die zu registrierende Nachricht einzutragen. In der Abbildung ist der Fall der Registrierung einer REQUEST-Nachricht dargestellt. Die innerhalb des ObjectServers ablaufenden Schritte zur Registrierung einer REQUEST-Nachricht sind in Abbildung 4.16 dargestellt.

Innerhalb des ObjectServers wird zunächst ein eindeutiger User-Process-Infoblock (UPI) für den aufrufenden Server gesucht und falls nicht vorhanden, neu erzeugt. Die Definition dieses Blocks ist in Abbildung 4.10 als `t_UserProcessInfo` dargestellt. Falls der ObjectServer zuvor noch keine Verbindung zum Kommunikationskanal (Channel) des Server-Receive-Thread des aufrufenden Servers hatte, wird nun diese Verbindung erstmals erstellt. Anschließend wird geprüft, ob bereits eine Registrierung für den übermittelten Hashindex besteht. Falls nicht, so werden die Listen innerhalb des ObjectServers um einen entsprechenden Eintrag erweitert und der aufrufende Server positiv benachrichtigt. Falls es doch schon einen identischen Hashwert-Eintrag bei gleichem Nachrichtennamen gibt, so wird anhand der Initialisierungsparameter entschieden, ob der Registrierungsantrag abgelehnt oder die alte Registrierung eines anderen Servers überschrieben wird. Generell kann es innerhalb der Hashindex-Berechnung in seltenen Fällen zu Überschneidungen kommen, da der Wertevorrat der Hashwertberechnung limitiert ist. In solchen Fällen muss der Nachrichtenname des Servers anders gewählt und der Vorgang erneut gestartet werden.

Wieder zurück in Abbildung 4.15 wird im Server auf eine erfolgreiche Registrierung der Nachricht geprüft und anschließend die Datenzeiger der Anwendung gesetzt.

Nachdem wie zuvor beschrieben innerhalb der Applikations-Clients und der Applikations-Server Nachrichten erfolgreich erzeugt bzw. registriert wurden, können die Prozesse nun die Vermittlungstätigkeit von MiRPA-X in Anspruch nehmen, um Nach-

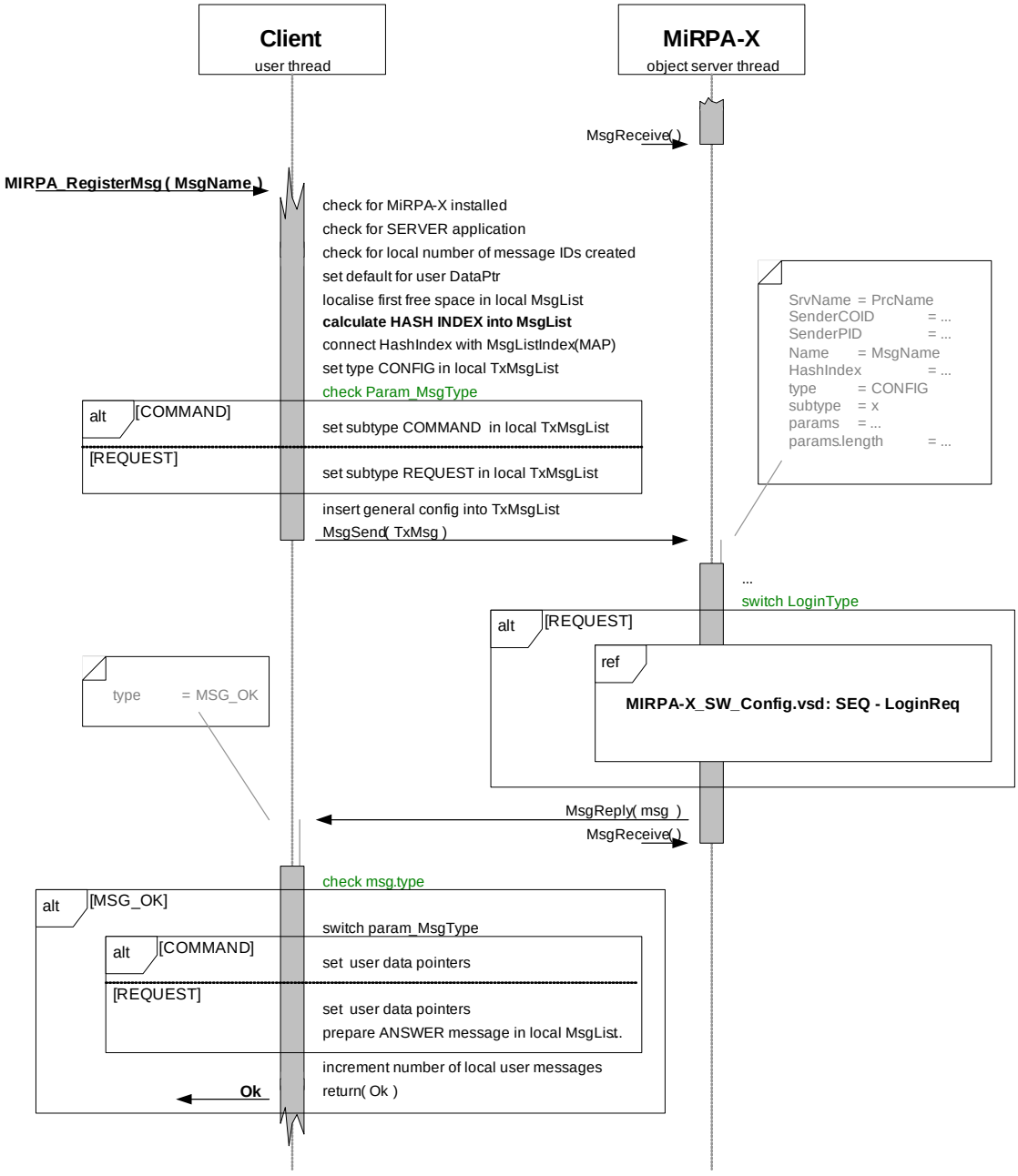



Abbildung 4.15: Vorgänge beim Registrieren eines Nachrichtenobjekts

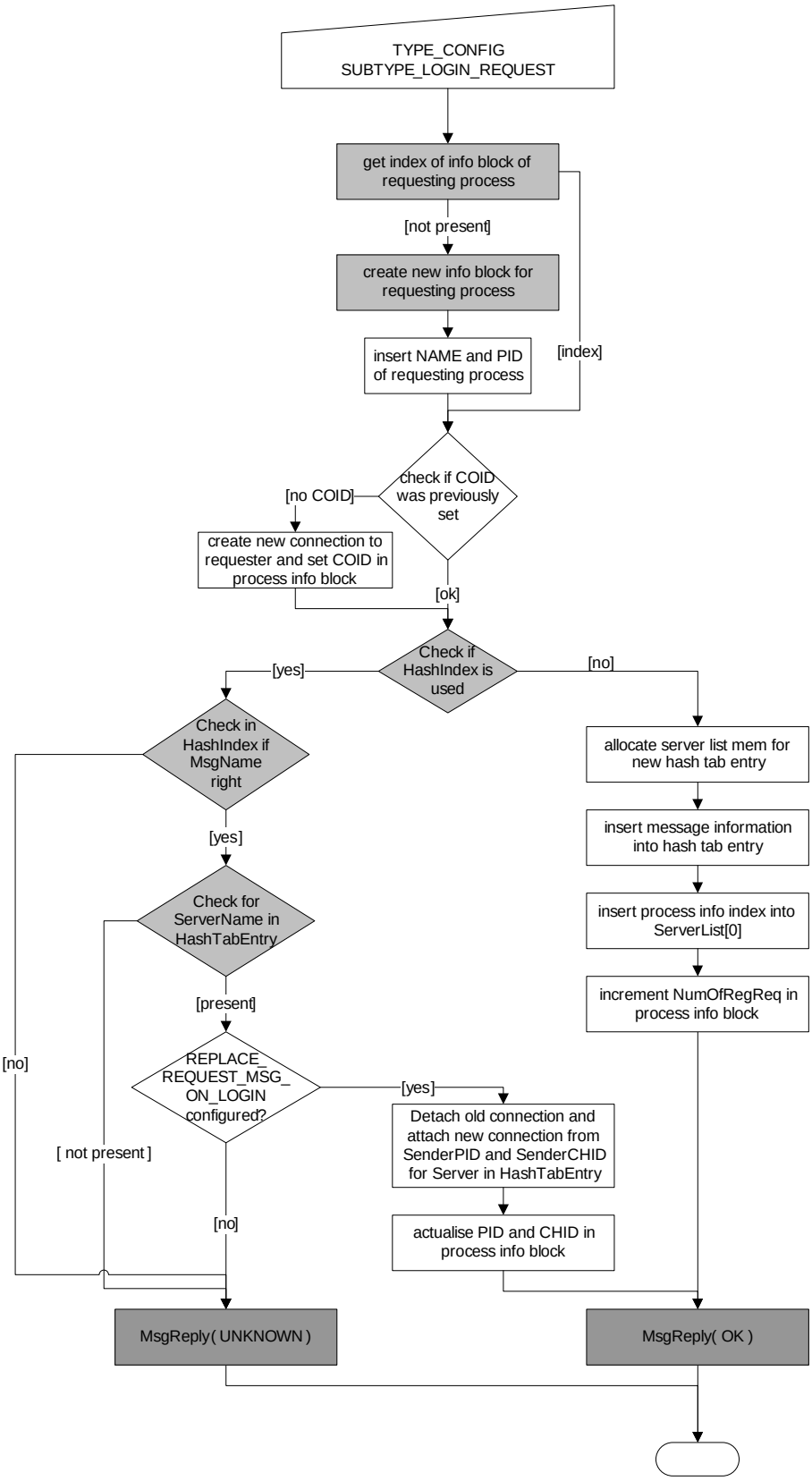



Abbildung 4.16: MIRPA-X_SW_Config.vsd, FLOW - LoginReq

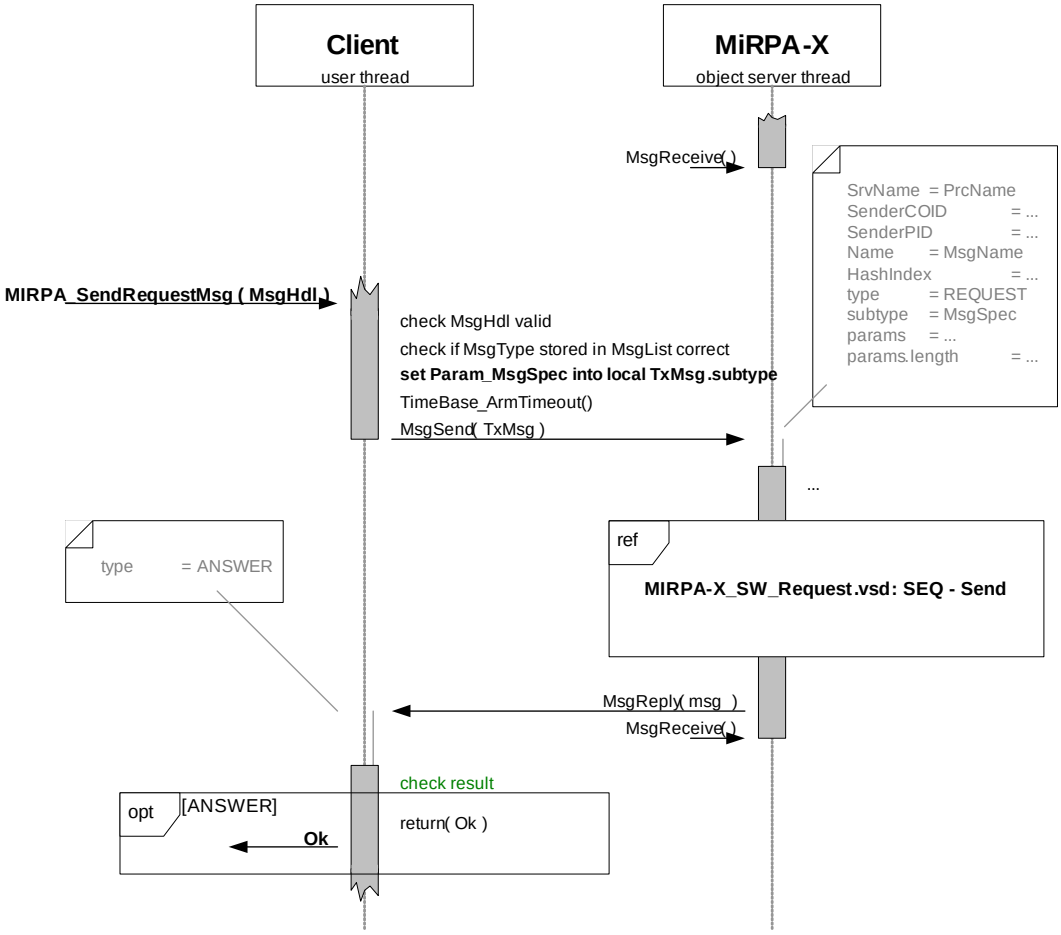



Abbildung 4.17: MIRPA-X_SW_API_Message.vsd, SEQ - SendReq

richten zu übertragen. Abbildung 4.17 zeigt dazu exemplarisch die Vorgänge zum Versenden einer REQUEST-Nachricht.

Wird das Versenden einer REQUEST-Nachricht über die API gestartet, so erfolgt zunächst die Prüfung der Zugriffsberechtigung des lokalen Client-Prozesses. Nun wird der Übergabeparameter `uiMsgSpec`, mit der die zu versendende Nachricht numerisch spezifiziert wird, in die Nachrichtenstruktur eingetragen und ein Timeout-Wert gesetzt. Dieser Wert gibt an, wieviel Zeit verstreichen darf, bis der Vorgang für den REQUEST abgeschlossen sein muss. Konnte während dieser Zeit keine Antwort vom ObjectServer empfangen werden, so wird der Aufrufer über einen Fehlerwert benachrichtigt. Der Datenbereich einer erfolgreich empfangenen ANSWER-Nachricht gelangt durch die Verwendung der vorgestellten Sende- und Empfangsstruktur direkt in den Speicherbereich, auf den der entsprechende Datenzeiger bereits zeigt. Alte Inhalte werden dabei überschrieben.

In Abbildung 4.18 sind die Vorgänge innerhalb des ObjectServers zur Vermittlung einer REQUEST- bzw. einer ANSWER-Nachricht dargestellt. Ausgangspunkt ist die über die entsprechende API-Funktion eingeleitete Ausführung der `MsgSend()`-Funktion des lokalen Client-Prozesses. Die Nachricht kommt im ObjectServer an und

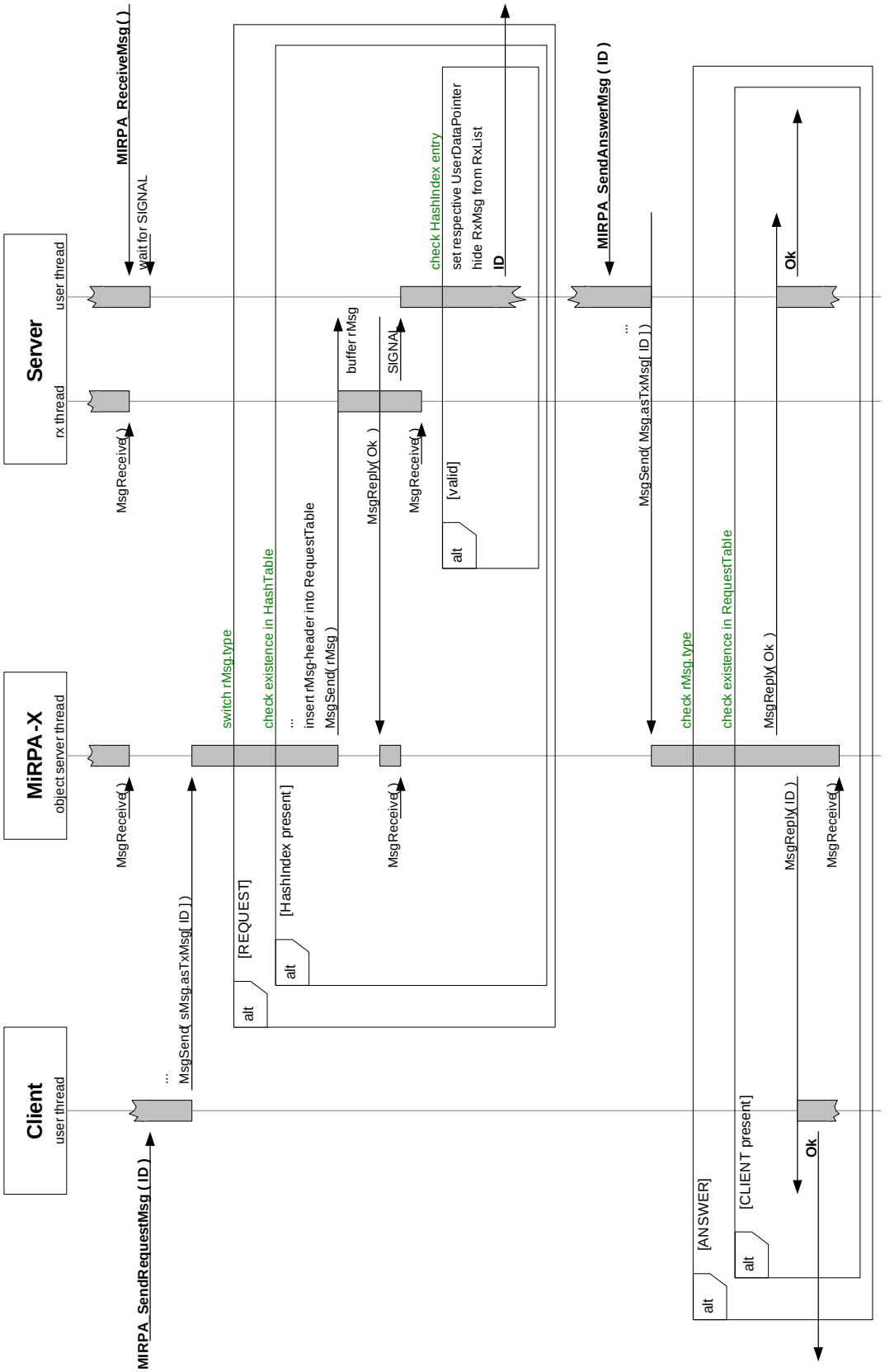



Abbildung 4.18: MIRPA-X_SW_Request.vsd, SEQ - Send

wird, sofern es sich um eine REQUEST-Nachricht handelt, anhand ihres bereits berechneten und mitgesendeten Hashwertes mit im ObjectServer bekannten Nachrichten verglichen. Ist die Nachricht bekannt, so erfolgt eine Suche nach dem zuständigen Server. Der Nachrichtenkopf wird in einer Liste mit offenen Anfragen gespeichert, um später beim Eintreffen der ANSWER-Nachricht eine korrekte Zuordnung zum anfragenden Client machen zu können. Anschließend erfolgt das Versenden der empfangenen Nachricht.

Der Server-Prozess besteht zu diesem Zeitpunkt aus mindestens zwei Threads, von denen der Server-Receive-Thread permanent auf einkommende Nachrichten wartet und diese beim Eintreffen puffert. Im User-Thread wurde zuvor die API-Funktion zum Empfangen einer Nachricht ausgeführt. Beide Threads sind somit auf den Empfang einer Nachricht bzw. eines Signals blockiert. Sobald die REQUEST-Nachricht vom ObjectServer eintrifft, wird diese vom Server-Receive-Thread gepuffert und sofort eine Bestätigungsnachricht an den ObjectServer verschickt. Dem User-Thread wird signalisiert, dass neue Nachrichten eingetroffen sind. In der API-Funktion wird nun geprüft, ob der Hashwert-Eintrag in der Nachricht lokal bekannt ist (der Server diese Nachricht also tatsächlich registriert hat). Ist dies der Fall, wird der Datenzeiger der Applikation auf den aktuellen Puffereintrag gesetzt, womit die Daten der Anwendung augenblicklich zur Verfügung stehen. Durch Rückgabe der lokalen Identifikationsnummer (sie entspricht derjenigen aus der Registrierungsfunktion) an den Anwender ist diesem bekannt, welche Nachricht angekommen ist.

Nachdem die empfangene REQUEST-Nachricht von der lokalen Server-Applikation ausgewertet und entsprechende Berechnungen durchgeführt wurden, erfolgt das Versenden einer ANSWER-Nachricht an den ObjectServer. Dies geschieht mit der abgebildeten API-Funktion `MIRPA_SendAnswerMsg()`, die analog zu der Sendefunktion aus dem Client (Abbildung 4.17) realisiert ist und deshalb hier nicht weiter beschrieben wird. Nach dem Eintreffen der ANSWER-Nachricht im ObjectServer wird zunächst der Nachrichtentyp überprüft. Sofern es sich um eine ANSWER-Nachricht handelt, durchsucht der ObjectServer die Liste mit offenen Anfragen nach einem passenden Empfängerprozess. Dann wird die Aktion an den Server-Prozess positiv bestätigt und die ANSWER-Nachricht an den entsprechenden Client-Prozess weitergeleitet, der daraufhin seine Sendefunktion verlässt.

### Vorgänge beim Shared-Memory-Zugriff

Gemeinsam genutzter Speicher spielt innerhalb der Verwendung der Middleware eine zentrale Rolle, da er sehr schnelle Kommunikations- und Synchronisationsmechanismen ermöglicht. Zunächst soll hier der generelle Umgang mit der Shared-Memory-Funktionalität von MiRPA-X erläutert werden, bevor in dem folgenden Abschnitt auf die Struktur und Verwendung von Synchronisationsmechanismen eingegangen wird.

Abbildung 4.19 zeigt den Vorgang zum Zugriff auf über MiRPA-X verwaltete Shared-Memory-Bereiche. Die dazu verwendete API-Funktion prüft zunächst, ob der gewünschte Shared-Memory-Bereich bereits lokal angefragt wurde und gibt bei Übereinstimmung der vorhandenen Parameter den alten Handle (ID) an den Aufrufer zurück.

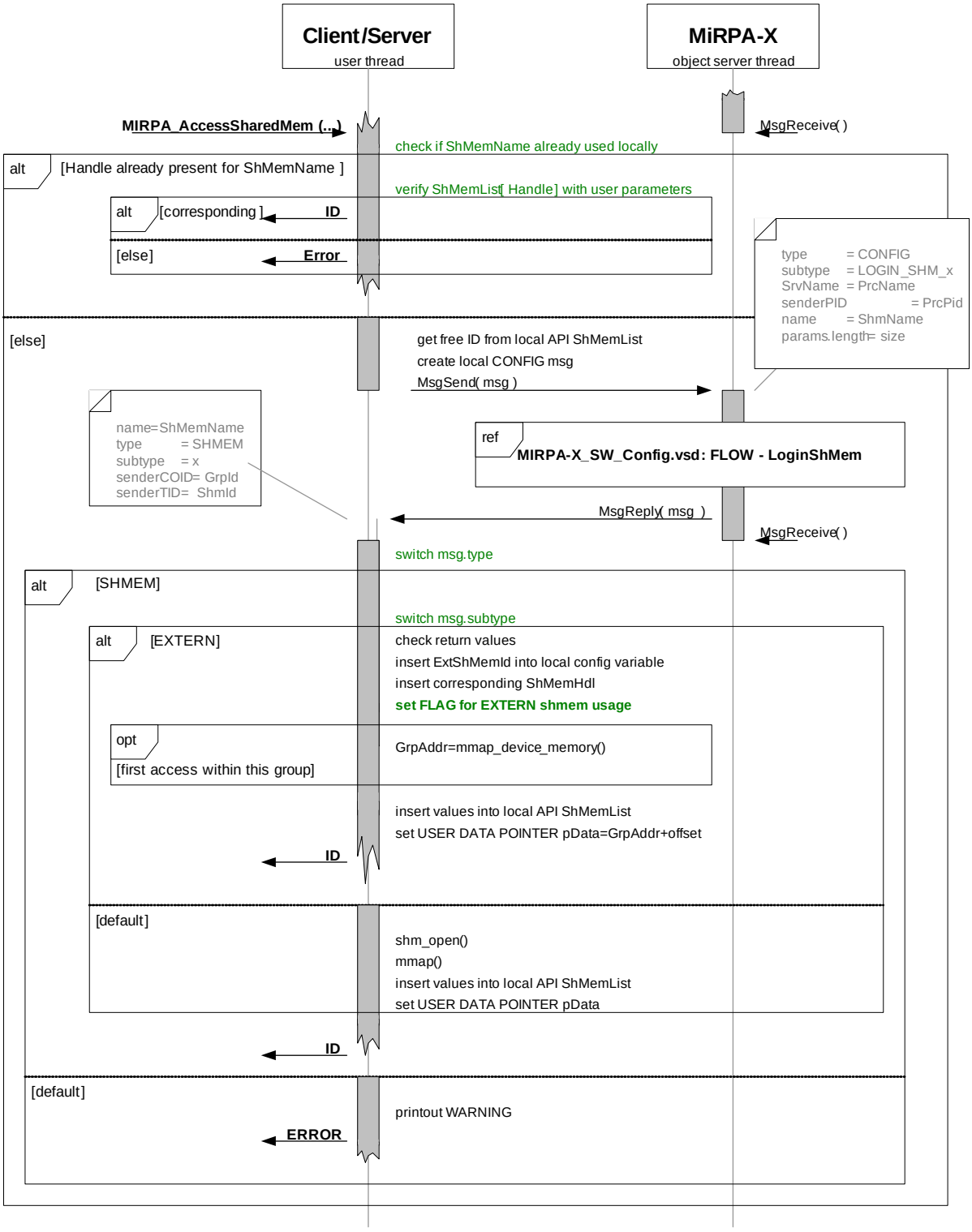



Abbildung 4.19: MIRPA-X_SW_API_ShMem.vsd, SEQ - AccessShMem

Falls der Shared-Memory-Bereich lokal noch nicht existiert, wird nach dem Generieren einer lokalen ID eine CONFIG-Nachricht mit den gewünschten Eigenschaften für den Shared-Memory-Bereich an den ObjectServer verschickt.

Abbildung 4.20 zeigt die Vorgänge innerhalb der Login-Funktion des ObjectServers, die notwendig sind, um einem anfragenden Prozess den Zugriff auf Shared-Memory zu gewähren. Hier wird zunächst der Prozess an sich über einen Process-Info-Block (PIB) registriert und dann überprüft, ob der angefragte Shared-Memory-Bereich bereits existiert oder neu erzeugt werden muss. Wenn er bereits existiert, wird der anfragende Prozess in die Liste der aktiven Zugreifer für den angefragten Shared-Memory-Block aufgenommen und dadurch eine Referenz zum entsprechenden PIB erzeugt. Für den Fall, dass es sich bei dem Typ des angefragten Speichers um „EXTERN“ handelt, wird dem anfragenden Prozess eine entsprechende Gruppen- und eine Extern-ShMem-Identifikationsnummer zurückgegeben, die weiter in der API verwendet wird, um einen automatischen Zugriff darauf zu koordinieren. Eine genauere Beschreibung dieser „EXTERN“-Funktionalität findet sich im folgenden Abschnitt.

Wenn der angefragte Shared-Memory-Bereich noch nicht existiert, muss die erste Prüfung auf den Typ „EXTERN“ negativ ausfallen, da es in diesem Fall vorbereitender Funktionalitäten bedarf, damit diese Art von Speicher sinnvoll genutzt werden kann. Bei negativem Vergleich wird der Shared-Memory-Bereich beim Betriebssystem angefragt und ein entsprechender ShMem-Info-Block (SIB) erzeugt, der dann dessen Eigenschaften speichert. In Abhängigkeit von dem angefragten Typ werden ein MUTEX und eine CONDITION VARIABLE für diesen Bereich erzeugt, um den Zugriff darauf zu koordinieren. Nun werden der PIB und der SIB miteinander verbunden und die API des Aufrufers erhält eine positive Bestätigung seiner Anfrage.

Wieder zurück in Abbildung 4.19 wird der innerhalb der empfangenen Nachricht übertragene Nachrichtentyp überprüft. Ist dieser „SHMEM“, so war die Registrierung innerhalb des ObjectServers erfolgreich und die weiteren Daten können aus der Nachricht entnommen werden. Handelte es sich dabei um „EXTERN“-Shared-Memory, so werden die übertragene Shared-Memory- und die Gruppen-Identifikationsnummer in die lokale Konfigurationsstruktur `t_ExtShMemList` eingetragen und ein Datenzugriff auf den entsprechenden Hardware-Speicheradressbereich vom Betriebssystem angefragt. Schließlich wird der Datenzeiger des Anwenders auf die entsprechende Datenspeicherposition gesetzt und die lokale Identifikationsnummer an den Anwender zurückgegeben. Dieser kann ab jetzt frei auf den Datenspeicher zugreifen.

### Struktur der EXTERN-Shared-Memory Funktionalität

MiRPA-X nutzt auch im Zusammenhang mit der Kommunikation über eine externe Hardwareschnittstelle Shared-Memory-Bereiche, um Daten dem Anwender zugänglich zu machen. In der Regel geschieht dies über die Standard-Funktionalität des Shared-Memory. Dabei wird in der Regel über ein externes Kommunikationsprotokoll (hier IAP, s.Abschnitt 4.3) eine Kopierfunktion realisiert, die externe Daten von dem Kommunikationsbus liest, in von MiRPA-X verwaltete Shared-Memory-Bereiche schreibt und damit die empfangenen Daten für alle Applikationsprozesse zugreifbar

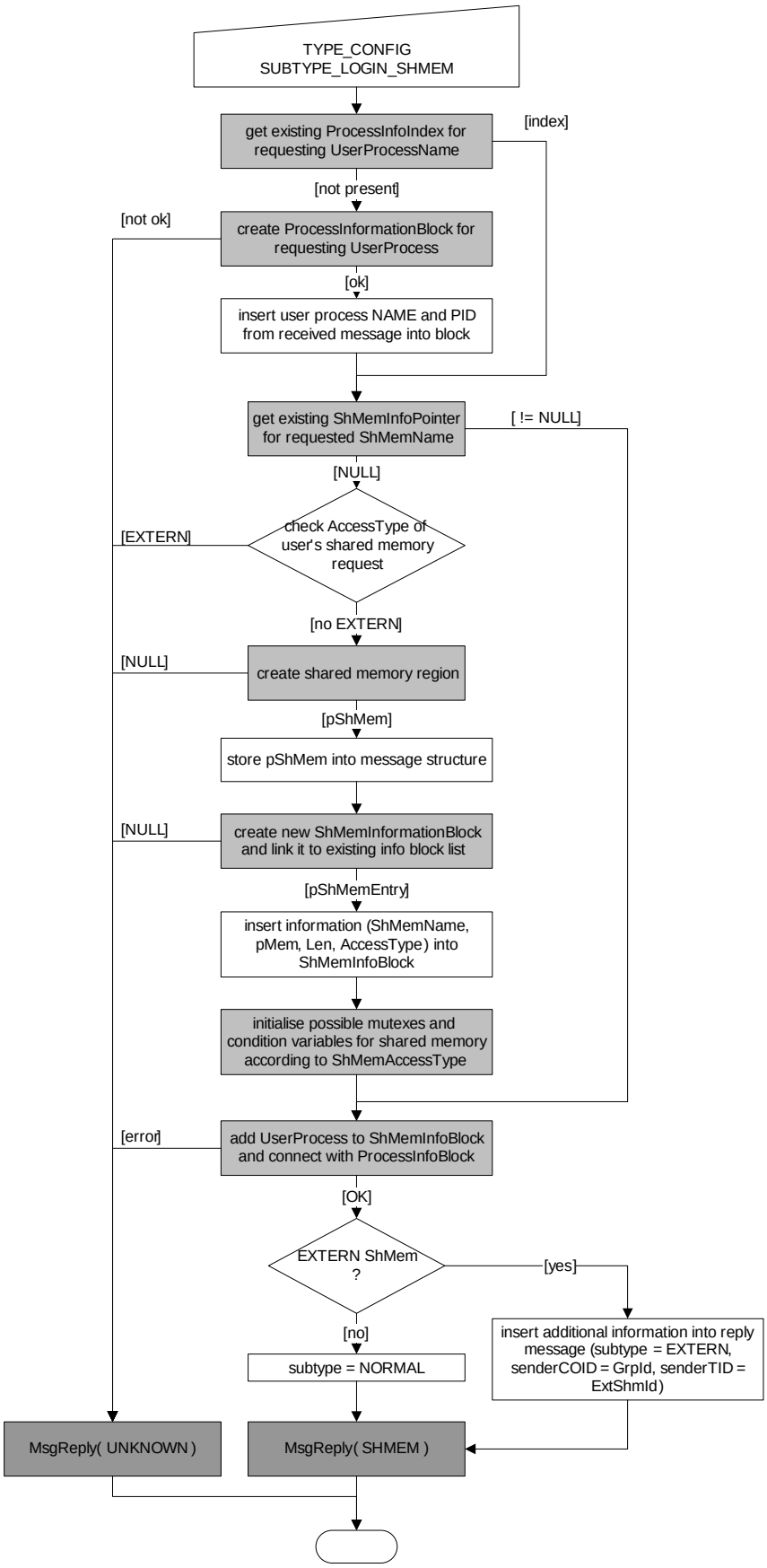



Abbildung 4.20: MIRPA-X_SW_Config.vsd, FLOW - LoginShMem

macht. Eine Sonderfunktionalität für Shared-Memory, die von MiRPA-X zur Verfügung gestellt wird, ist die „EXTERN“-Shared-Memory-Nutzung. Sie ermöglicht die effektive Ausnutzung des Direct-Memory-Access (DMA) einer externen Kommunikationsschnittstelle und kombiniert diese mit einem kopierfreien Datenzugriff für alle Anwendungsprozesse. Besonders für die Kommunikation großer Nachrichtenmengen ist die Verwendung dieser Funktionalität damit zu empfehlen.

Bei der standardmäßigen Shared-Memory-Nutzung im Zusammenhang mit externer Kommunikation wird für jedes übertragene Kommunikationstelegramm ein eigener Shared-Memory-Bereich bei MiRPA-X angemeldet. Sobald ein Telegramm empfangen ist, sorgt das entsprechende Kommunikationsprotokoll dafür, dass die Telegrammdaten in den dafür vorgesehenen Rohdaten-Speicherbereich geschrieben (dies kann per DMA der Hardware geschehen) und Sender und Empfänger bestimmt werden. Daraufhin erfolgt das Übertragen der Daten in die Shared-Memory-Bereiche der Anwendungsprozesse hinein. Dieser Funktionsablauf ist zwar flexibel, jedoch bei großen Datenmengen pro Telegramm aufgrund der Kopiervorgänge relativ langsam. Variationen in der Reihenfolge empfangener Telegramme werden automatisch aufgelöst. Im Gegensatz zu diesem Standard-Ansatz verwendet EXTERN-Shared-Memory einen einzelnen Speicherbereich zum Empfang *aller* Telegramme als Block. Diese Eigenschaft kann genutzt werden, wenn von der Kommunikationshardware aus *eine* Signalisierung (i.d.R. ein Interrupt) pro Datenblock erfolgen kann, woraufhin der DMA-Controller sämtliche Daten (alle empfangenen Telegramme hintereinander) aus der Hardware in den Arbeitsspeicher transferiert.

Abbildung 4.21 zeigt die Struktur und Verwendung von EXTERN-Shared-Memory. Links unten ist die Grundkonfiguration des Rohdaten-Speicherbereichs dargestellt, wie sie vom DMA-Controller erwartet wird. `HW_ADDRESS` ist die Basisadresse, `SIZE` die Größe des nutzbaren Speicherbereichs hierfür und `BYTE_OFFSET` der Byte-Positionszähler innerhalb des Speicherbereichs. Die Idee ist nun, den gesamten Rohdatenspeicher als Shared-Memory im Sinne von MiRPA-X zu deklarieren und damit das zeitaufwändige aktive Kopieren der Daten zu umgehen.

Drei Schritte sind notwendig, damit jedem Anwendungsprozess ein virtuelles Stück dieses Datenspeichers zugänglich gemacht werden kann. Zuerst muss über die Funktion `MIRPA_HookExternSharedMem()` der DMA-Rohdatenspeicher als Shared-Memory-Bereich deklariert werden. Hierfür wird in MiRPA-X eine „Shared-Memory-Group“ definiert, die `GRP_SIZE` groß ist, intern über einen Offset `GRP_BYTE_OFFSET` adressiert wird und sämtliche Telegrammdaten enthalten kann (`GRP_SIZE`). In einem zweiten Schritt wird dieser Gruppenspeicher mittels der Funktion `MIRPA_RegisterExternSharedMem()` partitioniert, so dass festgelegt wird, welche Shared-Memory-Bereiche in der jeweiligen Gruppe Platz finden sollen. Schritt drei umfasst nun das endgültige Registrieren der einzelnen Speicherportionen für die Applikationsprozesse, welches in der bekannten Form mit dem Zusatz „EXTERN“ innerhalb der Funktionsparameter für `MIRPA_AccessSharedMem()` erfolgt.

Bei manchen Kommunikationssystemen, wie z.B. bei dem innerhalb dieses Projekts verwendeten FireWire (IEEE1394), ist die Reihenfolge der Telegrammkommunikation nicht statisch vorgegeben, sondern richtet sich nach physikalischen Gegebenheiten (Entfernung zum ROOT). Diese Eigenschaft des unterlagerten Kommunikationssys-

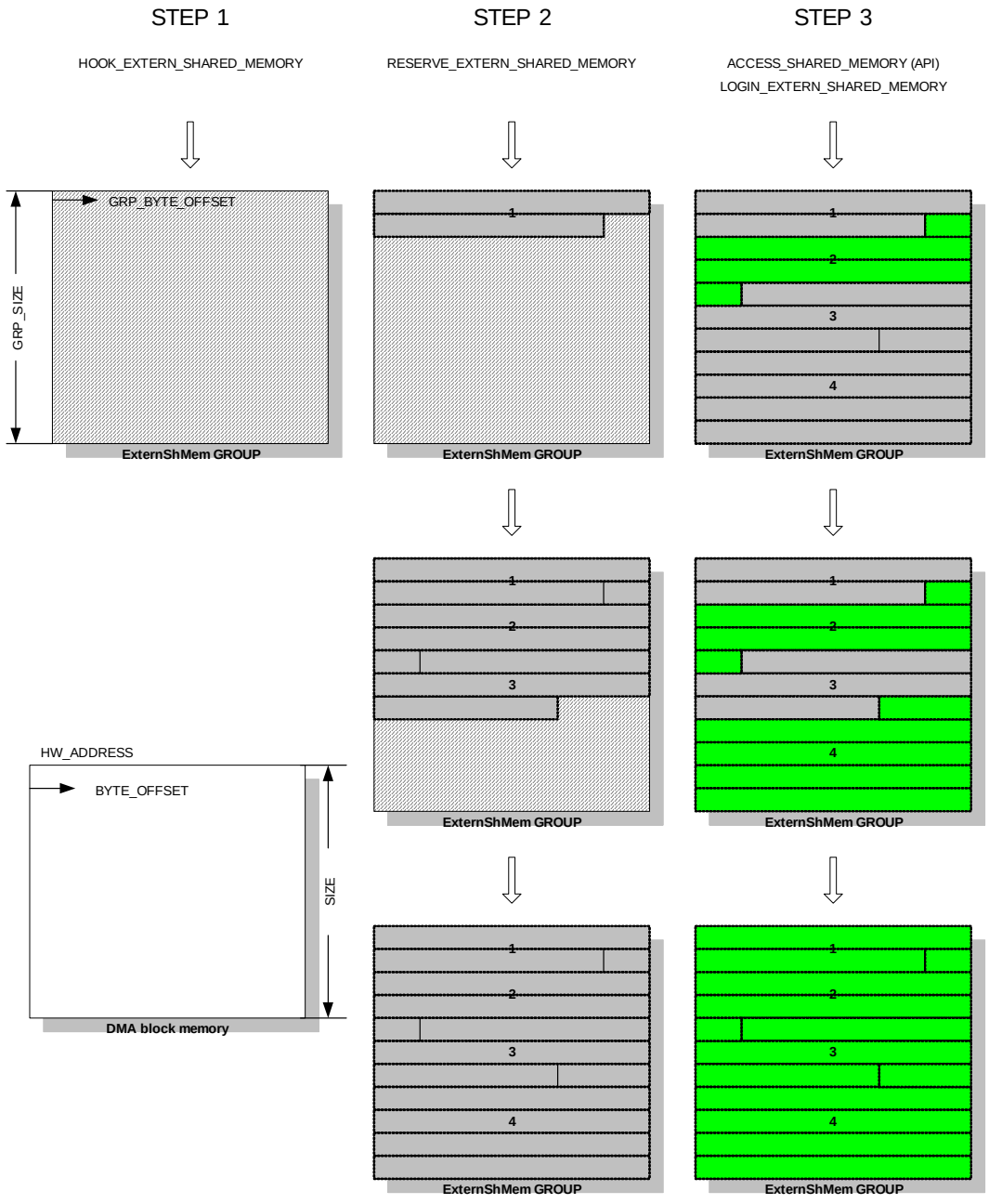



Abbildung 4.21: MIRPA-X_SW_Config.vsd, STRUCT - ExternShMem

tems bedingt die zunächst nachteilige Möglichkeit einer Reihenfolgenänderung der Speicherportionen innerhalb der Shared-Memory-Gruppe von Kommunikationszyklus zu Kommunikationszyklus und erfordert damit auch die dynamische Anpassung der jeweiligen Offsets `GRP_BYTE_OFFSET` der einzelnen Applikationen. Während der Konfigurierung einer Shared-Memory-Gruppe wird dazu ein zusätzlicher Shared-Memory-Bereich bei MiRPA-X angemeldet, der die `GRP_BYTE_OFFSET`-Werte für alle Applikationsprozesse enthält und aus der API-Ebene aller Prozesse zugegriffen werden kann.

Abbildung 4.22 zeigt die Struktur und den Ablauf (interne Vorgänge) der EXTERN-Shared-Memory-Funktionalität im laufenden Kommunikationszyklus. Ein gedachter Protokollprozess (wird später durch einen realen Prozess ersetzt) ruft dazu die Funktionen zum Konfigurieren und Steuern der EXTERN-Shared-Memory-Nutzung auf, die im Folgenden erläutert werden. Der Grundzustand ist hierbei, dass bereits alle für den aktuellen Berechnungszyklus erwarteten/erforderlichen Datentelegramme im Gruppenspeicher (in noch unbekannter Reihenfolge) vorliegen.

Zunächst werden, beginnend mit dem ersten Telegramm im Speicher, alle Telegrammköpfe nacheinander eingelesen (1) und dabei jeweils der Sender (Knoten-Identifikationsnummer) des Telegramms identifiziert. Je nach Knoten-ID wird nun über eine eindeutig zugeordnete `ExtShMemId` der entsprechende Offset für den Datenbereich des jeweiligen Telegramms unter der Strukturvariable `OffsetForId` innerhalb der Struktur `t_ExternShMemGroupInfo` eingetragen (2). Damit ist der Vorbereitungsvorgang abgeschlossen und jeder EXTERN-Shared-Memory-Zugriff ist für den aktuellen Berechnungszyklus eindeutig und korrekt festgelegt. Durch ein `MIRPA_ReleaseToken()`-Aufruf des Protokollprozesses (3) werden nun alle Token-Threads der Applikationsprozesse nach einander aufgerufen und greifen dabei auf die innerhalb der Telegramme übertragenen Daten externer Geräte zu (4). Dabei wird, nachdem der Token in einem Applikationsprozess lokal empfangen wurde (4.1), innerhalb der API-Schicht der Datenzeiger der Applikation für den EXTERN-Shared-Memory-Zugriff mittels des in der entsprechenden Struktur `t_ExternShMemGroupInfo` aktualisiert (4.2). Damit weist der Datenzeiger der Applikation direkt auf die entsprechenden Telegrammdaten. Die Applikation führt ihrerseits nun die vorgesehenen Operationen auf dem Datenspeicher aus (i.d.R. lesend) und gibt anschließend den Token über `MIRPA_ReleaseToken()` wieder an den SchedulerThread zurück (4.3). Dieser Vorgang wird so lange wiederholt, bis alle aktiven Token-Threads für diesen Echtzeitzyklus abgelaufen sind (5).

#### Vorgänge bei der Prozess-SYNCHRONISATION

In komplexen Steuerungssystemen spielt die Prozesssynchronisation eine entscheidende Rolle, um kooperierende Funktionalitäten echtzeitfähig zu koordinieren. MiRPA-X stellt zu diesem Zweck Mechanismen zur Verfügung, die mit Hilfe von Shared-Memory umgesetzt sind.

Abbildung 4.23 zeigt als Beispiel die API-Vorgänge zur Registrierung einer CONDITION-Variable für die Prozesssynchronisation. Nach der erfolgreichen Ermittlung einer freien lokalen Condition-Identifikationsnummer wird über einen API-internen Auf-

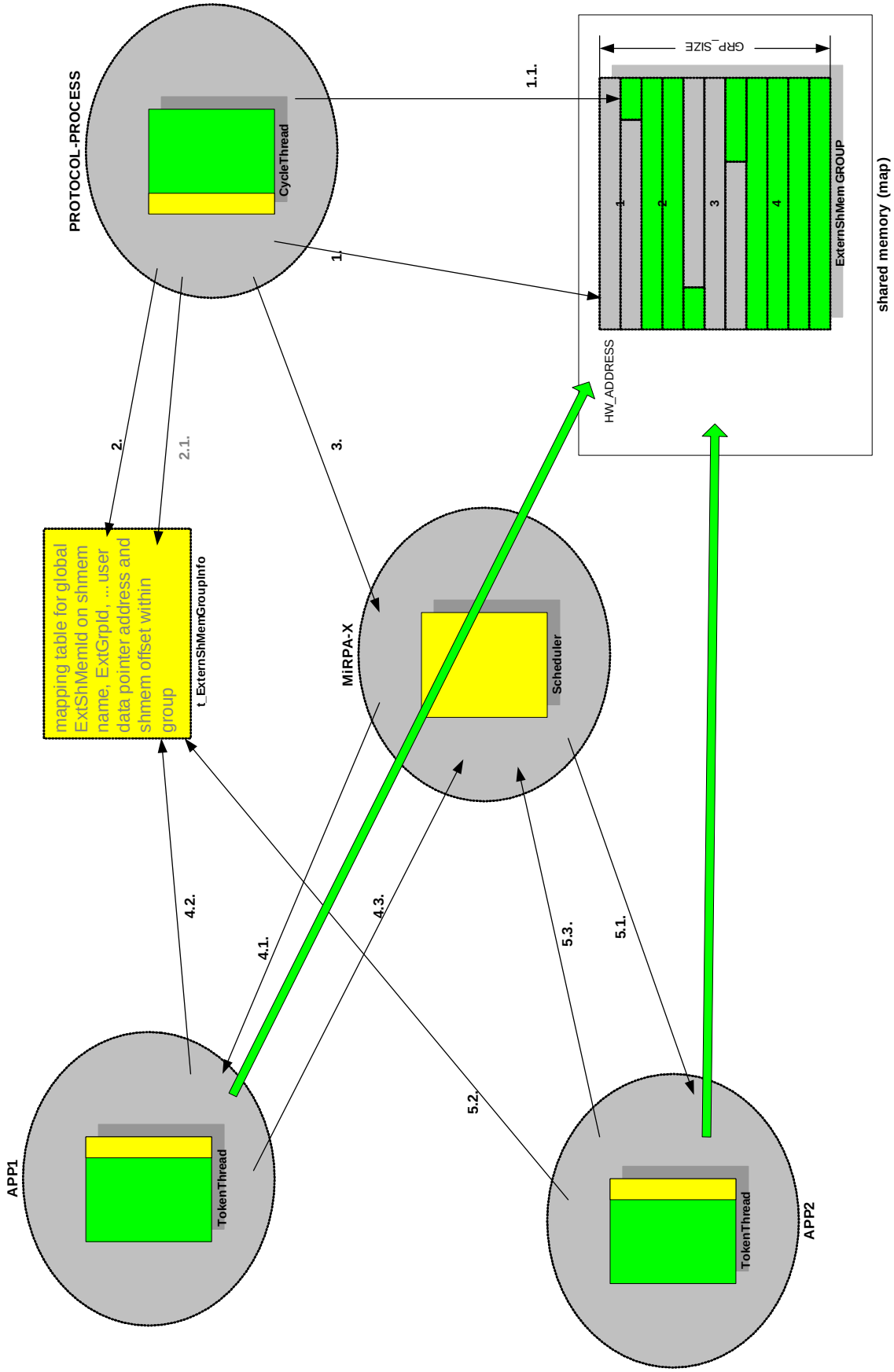



Abbildung 4.22: MIRPA-X_SW_API_ShMem.vsd, STRUCT - ExternConfig (3)

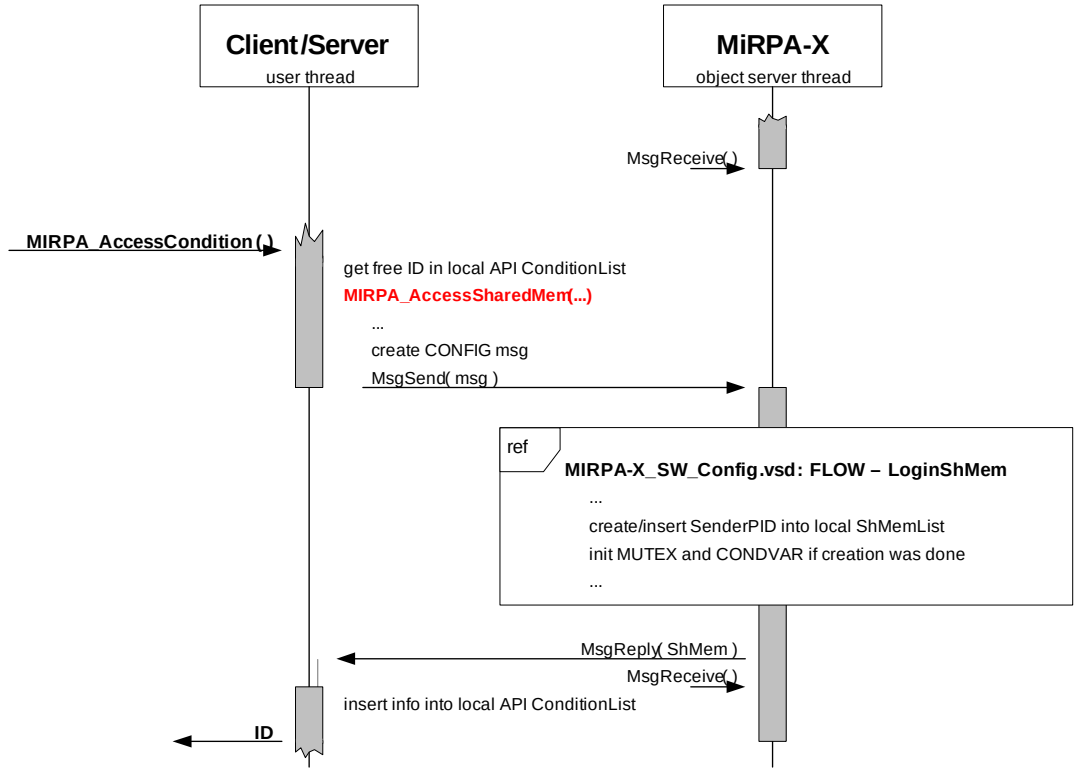



Abbildung 4.23: MIRPA-X_SW_API_ConditionVariable.vsd, SEQ - Access

ruf der API-Funktion `MIRPA_AccessSharedMem()`, wie weiter oben beschrieben, die ObjectServer-Funktionalität zur Erzeugung und Registrierung eines Shared-Memory-Bereichs durchgeführt. Ein Unterschied zur Standard-Registrierung ist hierbei, dass zusätzlich zum Shared-Memory-Bereich auch noch ein MUTEX und eine CONDVAR für diesen Bereich vom Betriebssystem angefordert werden. Nach dem Rücksprung in die API und die Beendigung der Funktion, kann die lokal zurückgegebene Identifikationsnummer dazu verwendet werden, den registrierten Synchronisierungsmechanismus zu nutzen.

Um den CONDITION-Mechanismus zu nutzen, werden zwei Synchronisationspartner nun gemeinsam betrachtet. Abbildung 4.24 zeigt dazu die Abläufe zwischen einem wartenden „Consumer“- und einem signalisierenden „Producer“-Prozess auf der API-Ebene. Aufgrund der verwendeten Darstellungsart können nicht alle möglichen Ablaufkonstellationen im Zusammenspiel zweier oder weiterer miteinander in Synchronisationsbeziehung stehender Prozesse beschrieben werden. Daher beschränkt sich die Darstellung auf den üblichen Fall, wobei natürlich alle weiteren Fälle innerhalb der Implementierung ebenfalls abgedeckt sind.

Nach Aufruf der Funktion `MIRPA_WaitForCondition()`, bei dem spezifiziert wird, auf welche Bedingung innerhalb der gewählten Synchronisationsvariable gewartet werden soll, wird zunächst geprüft, ob die Synchronisationsvariable überhaupt noch gültig ist. Die Gültigkeit einer Synchronisationsvariable kann bei Beendigung eines Zugriffs darauf mittels `MIRPA_ReleaseCondition()` bzw. `MIRPA_ReleaseConditionSilent()` beeinflusst werden. Je nach Wahl wird dabei der Wert der CONDITION auf `ABORT`

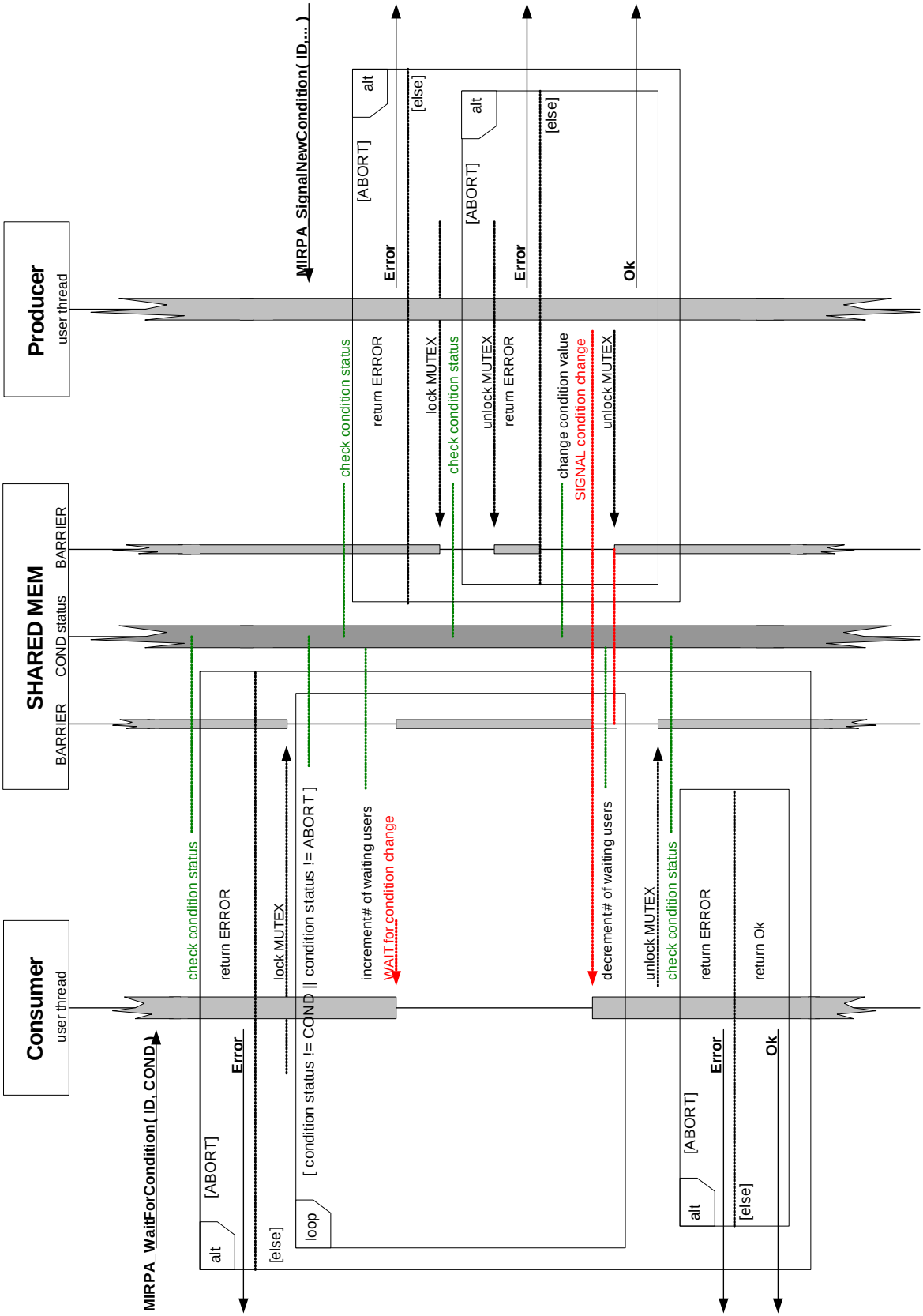



Abbildung 4.24: MIRPA-X_SW_API_ConditionVariable.vsd, SEQ - Sync

gesetzt oder nicht. Ist die Synchronisationsvariable also gültig, so wird nun, abgesichert über ein Mutex, innerhalb einer Schleife so lange auf die gewünschte Bedingung geprüft, bis diese eingetreten ist oder aber ein `ABORT` gesetzt wurde. Zum Zweck der Organisation wird die Anzahl der momentan auf eine spezifische Bedingung dieser Synchronisationsvariable wartenden Prozesse festgehalten. Damit lassen sich die Synchronisierungsmechanismen des Betriebssystems später weiter optimieren. Um nun den Prozessor nicht durch permanentes Polling zu belasten, wird der wartende Consumerprozess auf das Eintreffen eines Zustandswechsels blockiert. Dieser Zustandswechsel wird bei entsprechender Anregung über das Betriebssystem signalisiert.

Ein Producerprozess, der die Synchronisationsvariable verwendet um anzuzeigen, dass z.B. eine Datenberechnung erfolgt ist und nun eine weitere Auswertung geschehen kann, prüft innerhalb der API-Funktion `MIRPA_SignalNewCondition()` ebenfalls, ob die Synchronisationsvariable gültig ist und setzt diese dann, geschützt durch einen Mutex-Zugriff, auf den gewünschten Wert. Nun erfolgt die Signalisierung über das Betriebssystem, dass ein Zustandswechsel stattgefunden hat, welches so den blockierten Consumerprozess aufweckt. Dieser dekrementiert die Anzahl wartender Prozesse entsprechend und prüft erneut, ob die nun vorgefundene Bedingung der gewünschten Bedingung entspricht. Ist das der Fall, so gibt er den Mutex wieder frei und verfährt weiter im eigenen Programmcode. Damit während des blockierten Zustands des Consumerprozesses ein Bedingungswechsel innerhalb der Synchronisationsvariable stattfinden kann, sorgt das Betriebssystem dafür, dass ab dem Blockierungszustand bis zum Bedingungswechsel-Signal der Mutex wieder freigegeben ist, so dass ein schreibender Zugriff eines anderen Prozesses, geschützt durch Mutex, geschehen kann.

### Vorgänge beim Echtzeit-SCHEDULING

Die Realisierung des Echtzeit-Scheduling basiert auf der Nutzung aller in diesem Abschnitt bisher vorgestellten Mechanismen (Nachrichtentransfer, Shared-Memory und Synchronisation). Eine Instanzierung erfolgt während des Starts des ObjectServers, welcher in diesem Abschnitt ab Seite 102 beschrieben ist. Abbildung 4.25 zeigt den Ablauf des Schedulers, der im Folgenden erläutert wird.

Das Erzeugen des Scheduler-Threads mündet zunächst in eine Initialisierungsphase, innerhalb derer der Scheduler Shared-Memory beim ObjectServer anfordert und diesen mit Defaultwerten initialisiert. Auf diese Weise haben alle API-Funktionen der Applikationsprozesse später die Möglichkeit, ebenfalls Scheduler-interne Daten zu lesen und zu modifizieren. Anschließend wird über das Erzeugen eines separaten Scheduler-Kommunikationskanals eine direkte Zugriffsmöglichkeit für Prozesse auf den Scheduler ermöglicht. Falls bereits eine Scheduler-Konfigurationsdatei existiert, wird sie nun ausgelesen und die Prozess-Ablaufreihenfolge zunächst einmal festgelegt. Damit ist der Scheduler operativ. Bezüglich des Umgangs mit dem Scheduler werden drei Zustandsmaschinen unterschieden: Die Zustandsmaschine für den `Scheduler` erfasst, ob der Scheduler selbst installiert und initialisiert ist, und ob er sich in einem operativen Zustand befindet; die Zustandsmaschine für den `Token` erfasst den aktuellen Zustand des zwischen den Token-Threads ausgetauschten Aktivierungssignals; die Zustandsmaschine für den `TokenThread` erfasst den individuellen Zustand einer

jeden vom Anwender bereitgestellten Echtzeit-Funktionalität innerhalb eines Applikationsprozesses.

Ist also der Scheduler aktiv (`Scheduler == RUNNING`), wird, falls der Token (noch) nicht im Umlauf ist (z.B. `Token == IDLE`), d.h. die Echtzeitfunktionalitäten innerhalb der Applikationsprozesse inaktiv sind, auf das Eintreffen eines Kommandos von einem Applikationsprozess gewartet. Die Kommandos werden von den Applikationsprozessen über die entsprechenden API-Funktionen per Nachricht an den Scheduler verschickt und dort ausgeführt (z.B. `MIRPA_ReInitToken()` zur Ausführung des Pfads für `REINIT`). Je nach Kommando erfolgt die Aktivität des Schedulers dann wie in der Abbildung dargestellt.



Sobald der Token über die Applikation im Zustand `Token == RUNNING` versetzt ist, wird das im ersten Schritt festgesetzte Kommando in einem zweiten Schritt in der Form ausgewertet, dass nun der Token-Thread-Ablauf umgesetzt wird. Hierbei kommt die während der Initialisierungsphase des Schedulers festgelegte oder später über API-Funktionen modifizierte Ablaufliste der Applikationen zum Einsatz. Je nach Listenelement, auf das über einen fortlaufenden `index` zugegriffen wird, erfolgt die Auswahl der zu startenden Echtzeitfunktionalität. Dabei erfolgt ein tatsächlicher Aufruf nur dann, wenn mindestens eine vorher per Konfiguration festgelegte Minimalanzahl an aktiven Token-Threads vorhanden ist.

Bevor in einem dritten Schritt der Token tatsächlich an einen Token-Thread verschickt wird, erfolgt eine Prüfung, ob sich der entsprechende Token-Thread auch im zum Empfang erforderlichen Aktivitätszustand befindet. Im Zustand `TokenThread == READY` wird der Token dann an den Thread abgeschickt, was diesen dann zur Ausführung bringt. Bei einem abweichenden Token-Thread-Zustand wird zusätzlich noch die spezifische und bei der Initialisierung festgelegte Rolle des Token-Threads innerhalb des Gesamt-Steuerungssystems berücksichtigt. Auf diese Weise kann z.B. eine systemübergreifende Zeitsynchronisation direkt innerhalb der Applikation mit Unterstützung der API abgewickelt werden.

Abbildung 4.26 und Abbildung 4.27 zeigen die möglichen Zustände für den `Token` und den `TokenThread` in Zusammenhang mit den API-Funktionen zu ihrer Beeinflussung. Während die API-Funktionen zur Beeinflussung des `Token`-Zustands von einem übergeordneten Applikationsprozess aufgerufen werden sollten, sind die API-Funktionen zur Beeinflussung des `TokenThread`-Zustands von allen Applikationsprozessen auszuführen, um damit den Funktionsablauf lokal zu koordinieren. Der Aufruf von `MIRPA_StopToken()` bewirkt, dass der Token nicht mehr weiter verschickt wird und weckt gleichzeitig alle wartenden Token-Threads auf, so dass lokal auf das Ausbleiben des Token reagiert werden kann.

### 4.2.3 Bestätigende Messungen

In Abschnitt 2.1.3 wurden die allgemeinen Voraussetzungen für Echtzeitfähigkeit von Softwaresystemen vorgestellt. An dieser Stelle erfolgt die Darstellung bestätigender Messungen, um die Echtzeitfähigkeit der beschriebenen Kommunikations-Infrastruk-

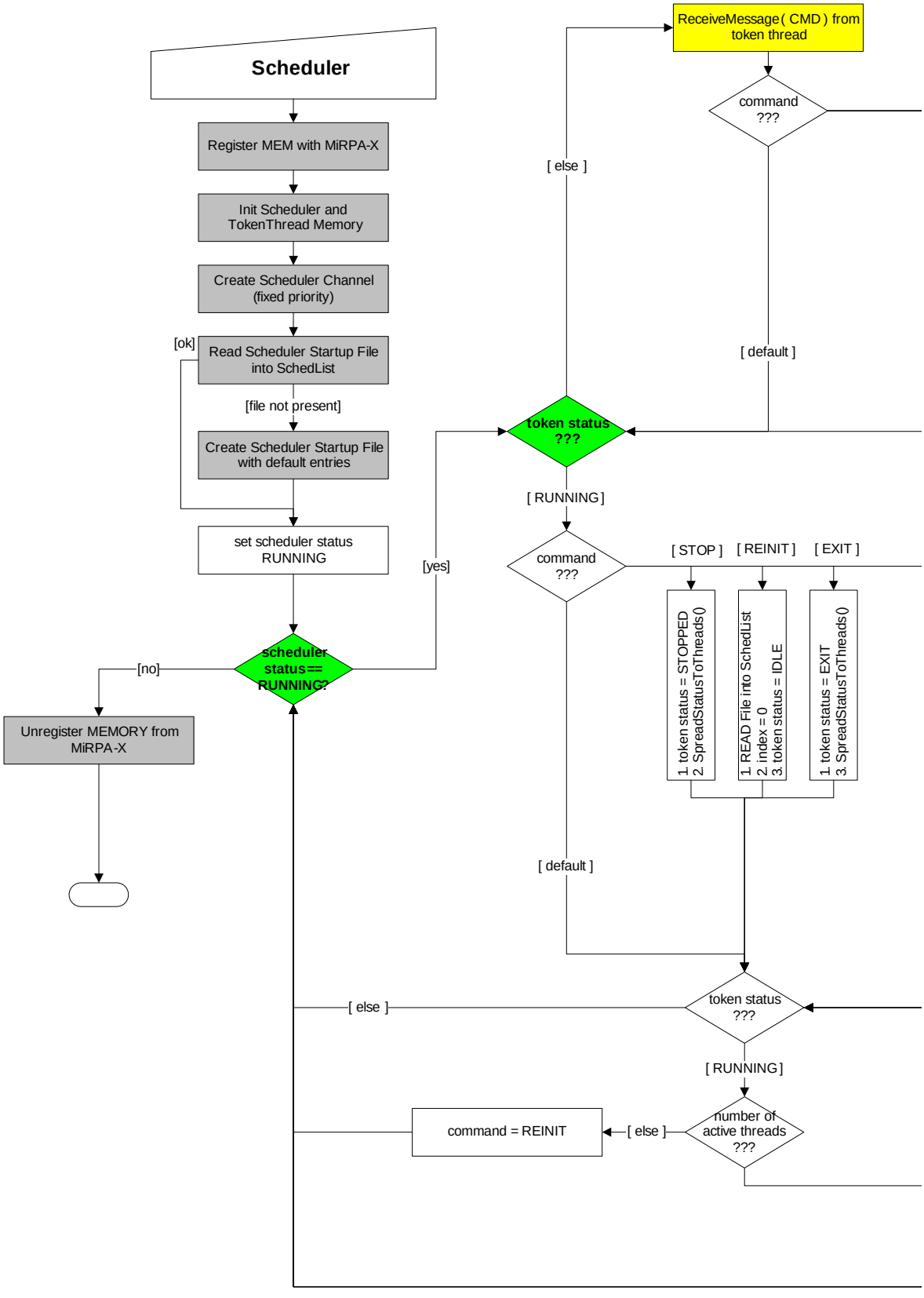
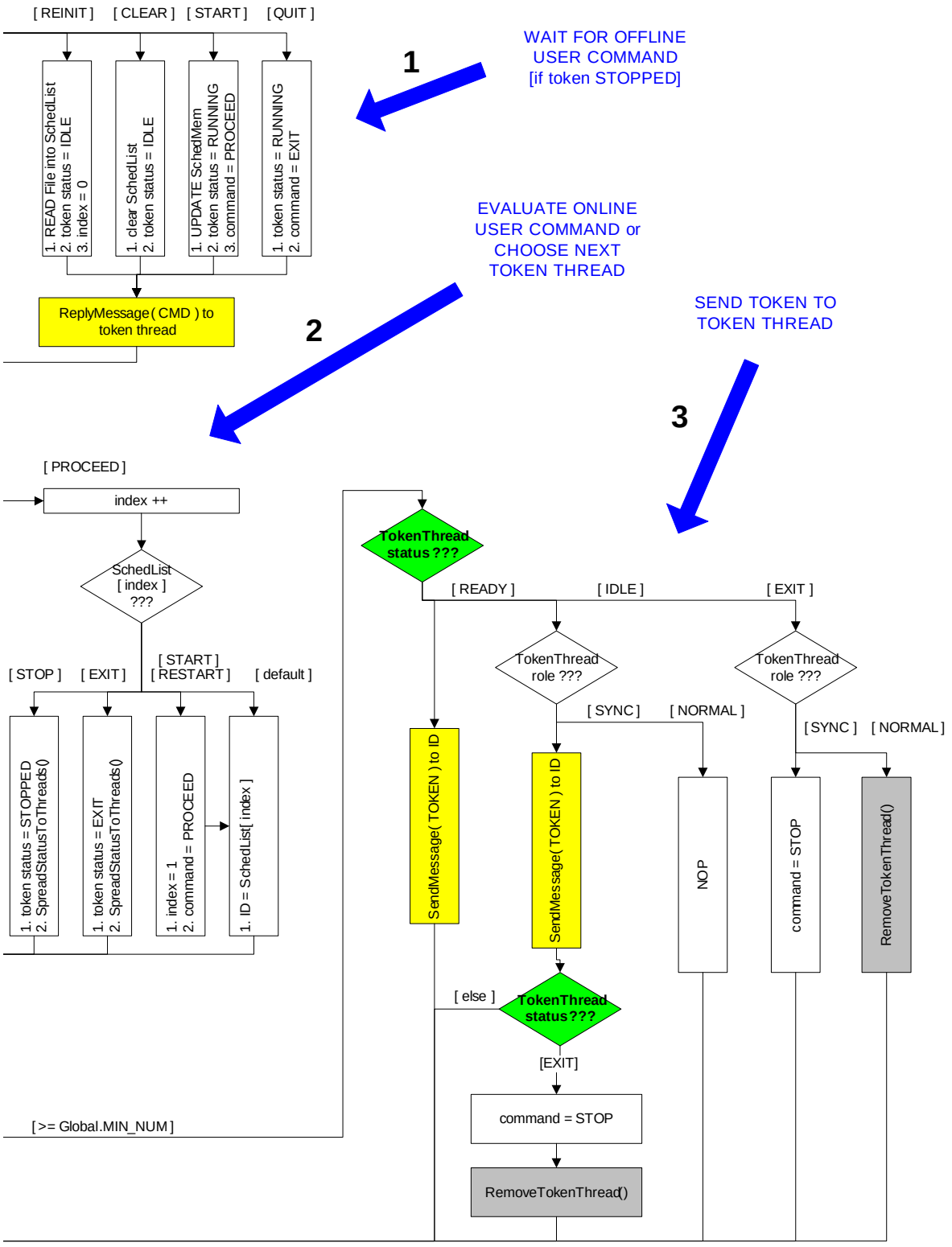



Abbildung 4.25: MIRPA-X_SW_Scheduler.vsd, FLOW - Control


[ REINIT ]
[ CLEAR ]
[ START ]
[ QUIT ]
1. READ File into SchedList
2. token status = IDLE
3. index = 0
1. clear SchedList
2. token status = IDLE
1. UPDATE SchedMem
2. token status = RUNNING
3. command = PROCEED
1. token status = RUNNING
2. command = EXIT
ReplyMessage ( CMD ) to token thread
1
WAIT FOR OFFLINE USER COMMAND [if token STOPPED]
2
EVALUATE ONLINE USER COMMAND or CHOOSE NEXT TOKEN THREAD
3
SEND TOKEN TO TOKEN THREAD
[ PROCEED ]
index ++
SchedList [ index ] ???
[ STOP ]
[ EXIT ]
[ START ] [ RESTART ]
[ default ]
1. token status = STOPPED
2. SpreadStatusToThreads()
1. token status = EXIT
2. SpreadStatusToThreads()
1. index = 1
2. command = PROCEED
1. ID = SchedList[ index ]
[ >= Global.MIN_NUM ]
TokenThread status ???
[ READY ]
[ IDLE ]
[ EXIT ]
TokenThread role ???
[ SYNC ]
[ NORMAL ]
TokenThread role ???
[ SYNC ]
[ NORMAL ]
SendMessage( TOKEN ) to ID
SendMessage( TOKEN ) to ID
NOP
command = STOP
RemoveTokenThread()
[ else ]
TokenThread status ???
[EXIT]
command = STOP
RemoveTokenThread()

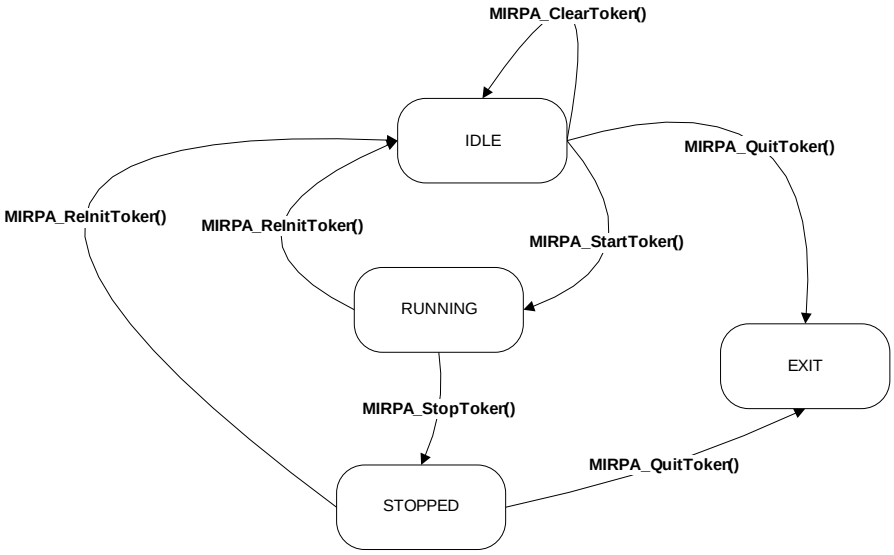



Abbildung 4.26: MIRPA-X_SW_Scheduler.vsd, STATE - Token

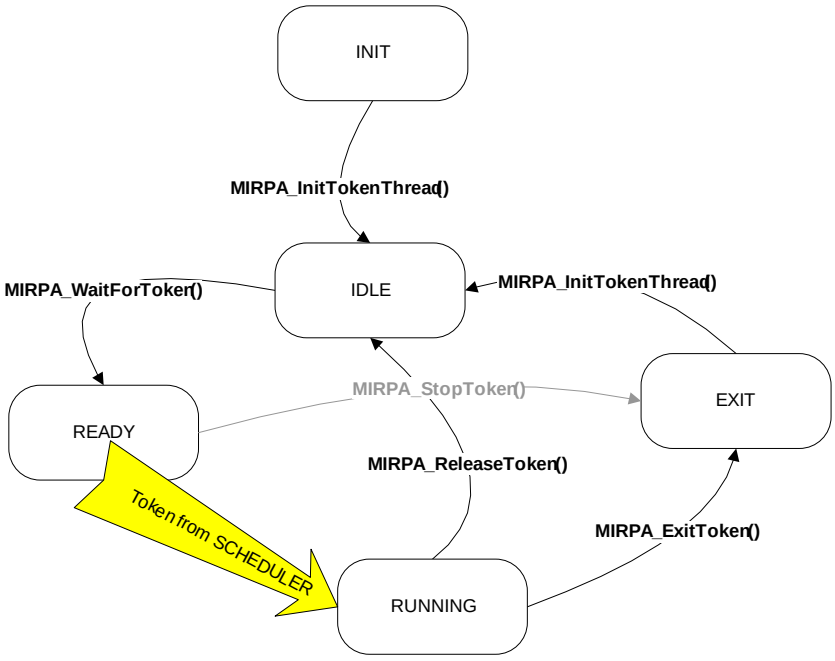



Abbildung 4.27: MIRPA-X_SW_Scheduler.vsd, STATE - TokenThread

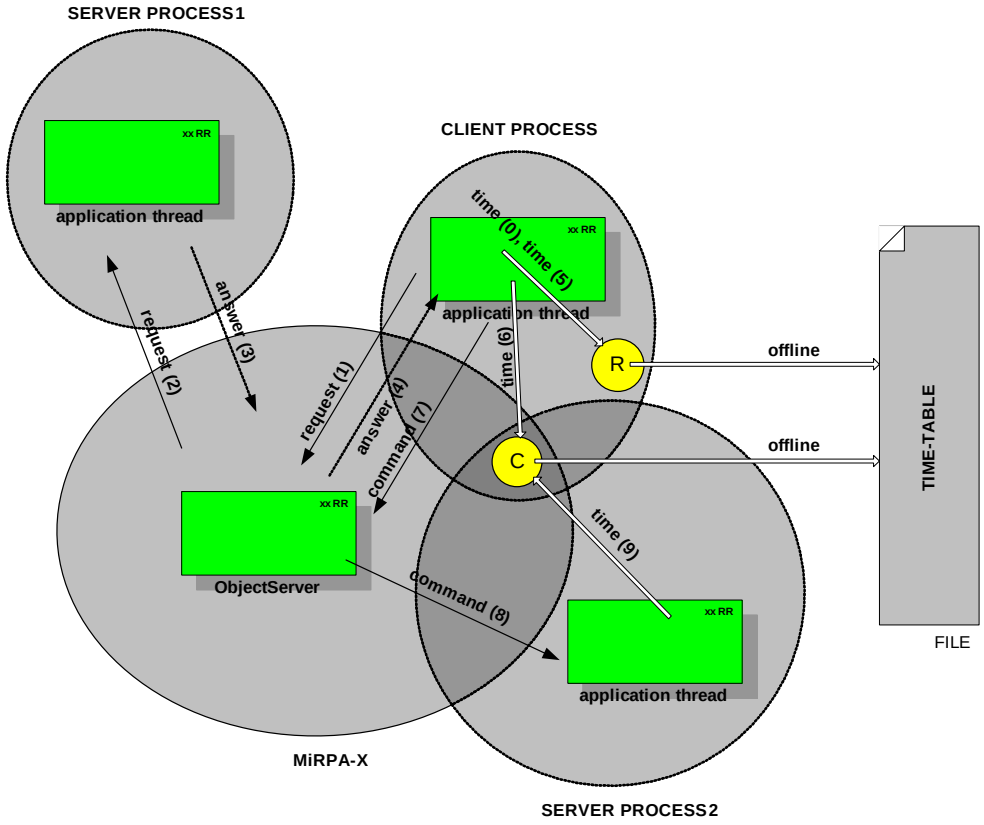



Abbildung 4.28: Aufbau der Messumgebung für Nachrichtentransfer über MiRPA-X ObjectServer

tur praktisch beurteilen zu können. Da alle mit MiRPA-X verbundenen Anwendungsprozesse nur die von MiRPA-X zur Verfügung gestellten Mechanismen für Datentransfer, Synchronisation und Ablaufsteuerung verwenden, und das auch nur in der zulässigen Weise, ist eine Untersuchung der MiRPA-X-Funktionalitäten hinreichend. Eine Beschreibung der Kommunikations-Infrastruktur als Ganzes erfolgt zusätzlich in Abschnitt 5.

Prinzipiell realisiert MiRPA-X ein Multilevel-Scheduling, bei der auf hoher (echtzeitkritischer) Prioritätsebene eine rein präemptiv arbeitende Ablaufsteuerung (FCFS) zur Verfügung steht. Auf niedrigerer Prioritätsebene wird durch Nutzung von RR-Scheduling eine faire Prozessoraufteilung realisiert. Zur Unterstützung der Synchronisationsmechanismen auch zwischen diesen beiden Ablaufebenen stellen Prioritätenvererbung und adaptives Scheduling minimale Latenzzeiten sicher. Der Nachrichtentransfer ist auf Übertragungsgeschwindigkeit optimiert worden; dies kommt vor allem dem Soft-Realtime-Bereich zugute.

### Nachrichtenkommunikation

Um die Übertragungseigenschaften der asynchronen Nachrichten-Kommunikation quantitativ zu bestimmen, wurde eine flexible Messumgebung entwickelt, deren schematischer Aufbau in Abbildung 4.28 dargestellt ist. Die Messumgebung besteht aus dem ObjectServer und parametrierbaren Client- und Server-Prozessen, die mit einer wählbaren Priorität im System ablaufen und konfigurierbare Nachrichten senden und empfangen können. Das Kernstück der Messfunktionalität bildet eine Funktionsbibliothek, die es ermöglicht, entweder die Ausgänge des Parallelports des PC oder den internen Systemtimer zu nutzen, um Vorgänge auf dem Rechner zeitlich festzuhalten und anschließend zueinander in Beziehung zu setzen. Dadurch, dass die Einstellungen der Messumgebung reproduzierbar sind, konnten auf diese Weise während der Entwicklung hilfreiche Hinweise zur Optimierung der Implementierung erhalten werden. Auch konnte durch vergleichbare Messungen auf unterschiedlicher Rechnerhardware die Stabilität und Zuverlässigkeit durch das Ergebnis immer qualitativ gleichen Systemverhaltens gezeigt werden.

In Abbildung 4.28 sind zwei charakteristische Messabläufe skizziert. Dabei wird im ersten Fall der Zyklus zur Vermessung von `REQUEST`-Nachrichten dargestellt (Index 0 bis 6). Der Client speichert zunächst die aktuelle Systemzeit und setzt unmittelbar danach eine Anfrage ab. Dieser `REQUEST` wird vom ObjectServer an den entsprechenden Server weitergeleitet und dort sofort beantwortet. Der ObjectServer erhält die Antwort (`ANSWER`), leitet sie an den Client weiter, welcher daraufhin wieder die Systemzeit in seinem Adressraum speichert. Im zweiten Fall wird der Zyklus zur Vermessung von `COMMAND`-Nachrichten dargestellt (Index 7 bis 9). Hier registriert der Client auch erst die aktuelle Systemzeit und setzt anschließend den `COMMAND` ab. Der ObjectServer leitet die Anfrage an den entsprechenden Server weiter, welcher sofort nach dem Empfang selbst die aktuelle Systemzeit in dem gleichen, gemeinsam genutzten Speicher ablegt.

Die Messzyklen können beliebig oft wiederholt werden, um verlässliche Aussagen über Mittelwerte und Abweichungen machen zu können. Die während der Messungen aufgenommenen Zeiten werden nach Abschluss der Messungen aus den Speichern ausgelesen und sortiert nach Konfiguration, Priorität und übertragener Datenmenge in Dateien abgelegt, die dann in Form von Kurven dargestellt werden können. Für die Messungen kam ein PC mit einem 2,4GHz Athlon Prozessor zum Einsatz.

Abbildung 4.29 zeigt das Ergebnis der zeitlichen Vermessung des Datentransfers für `COMMAND`- und `REQUEST`-Nachrichten. Als QNX-Prioritäten waren jeweils „15RR“ verwendet worden. Die obere Punktefolge (grün) zeigt den Zeitbedarf für `REQUEST`-Nachrichten, während die untere (blau), etwas schnellere Punktefolge den Zeitaufwand für `COMMAND`-Nachrichten verdeutlicht. Erst ab einer Nutzdatenmenge von ca. 50 Byte steigt der Zeitaufwand für übertragene Daten auf über 10 bzw. $8\mu s$. Der konstante Zeitabstand im Bereich von $2\mu s$ zwischen den beiden Kommunikationsarten erklärt sich aus dem zusätzlichen Signalisierungsaufwand zur Benachrichtigung des entsprechenden Servers.

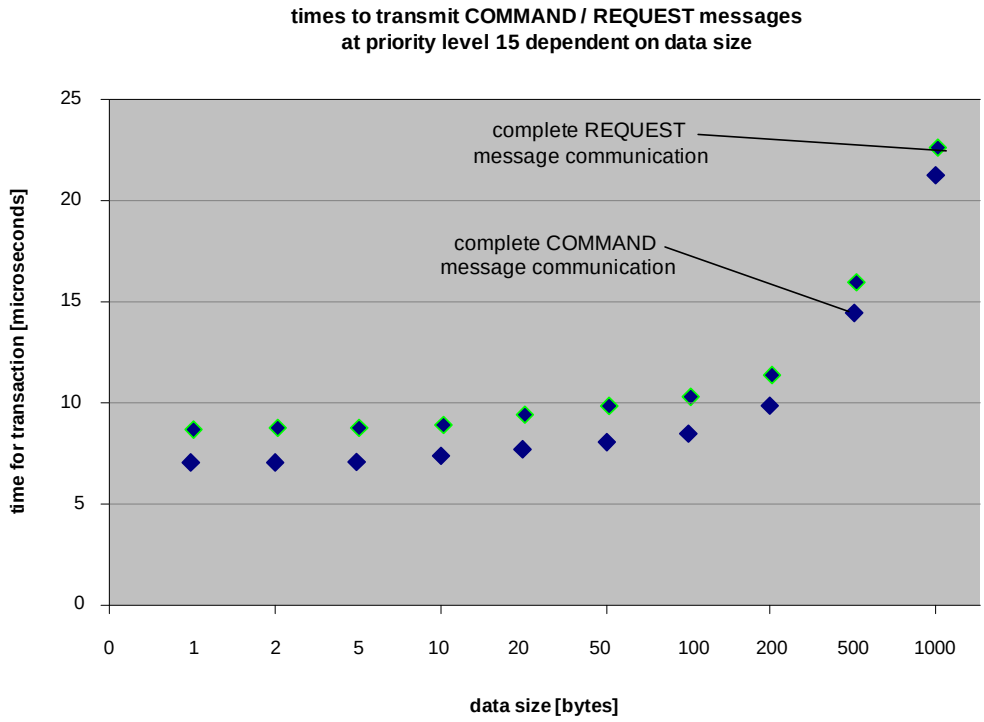



Abbildung 4.29: Übertragungszeiten für asynchrone Nachrichten

### Synchronisation und Scheduling

Zur Realisierung der Synchronisations- und Schedulingfunktionalitäten standen bei Verwendung des Betriebssystems QNX Neutrino mehrere Möglichkeiten zur Auswahl. Dabei wurde stets so verfahren, dass die Zeit zwischen dem Eintritt zweier aufeinander folgender `TOKEN-THREADS` gemessen wurde, die im konkreten Anwendungsfall mit Funktionalität gefüllt sind, zu Messzwecken aber leer gelassen wurden. Die unterschiedlichen Ergebnisse sind in Abbildung 4.30 dargestellt.

Die Verwendung von „ConditionVariables“ wird auf zwei unterschiedliche Weisen angeboten. Sie nutzt den unterlagerten Mutex-Mechanismus und zeigt bei ungünstiger Token-Thread-Verteilung einen von der Anzahl aktiver Token-Threads (RT-Threads) abhängigen Zeitbedarf (blaue, steigende Kurve). Eine bessere Möglichkeit zeigt die mittlere, dunkelgrüne Kurve. Hier wird die Auswahl der Token-Threads über eine aufwändigere Verarbeitung innerhalb des Schedulers vorgegeben. Die Verzögerungszeiten liegen bei $16\mu s$, unabhängig von der Anzahl der verwendeten RT-Threads. Die hellgrüne Kurve verdeutlicht die Zeitverhältnisse bei einer Signalisierung über leere Nachrichten (Message-Passing). Dies scheint der schnellste Signalisierungsmechanismus zu sein. Er liegt konstant bei ca. $4\mu s$ und ist damit für die Echtzeit-Ablaufsteuerung sehr gut geeignet.

Um die realisierten zeitlichen Eigenschaften in einem realen Steuerungssystem sicherzustellen, ist es nötig, alle hochprioren Prozessaktivitäten des Betriebssystems auszuschalten. Dazu gehören Einstellungen im BIOS (z.B. USB-Treiber). Sollten diese Einstellungen nicht vorgenommen werden, so kann es in regelmäßigen Abständen

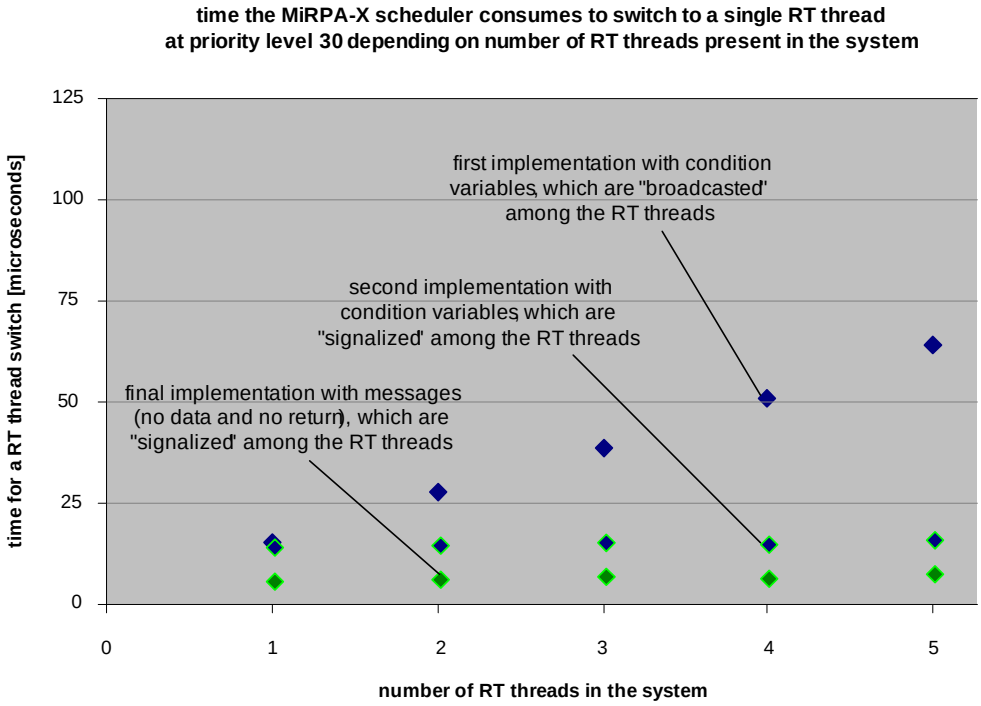



Abbildung 4.30: Schedulingzeiten bei unterschiedlichen Mechanismen

vorkommen, dass Echtzeitbedingungen verletzt werden, da zeitliche Vorgaben nicht zuverlässig eingehalten werden können.

## 4.3 Kommunikationsprotokoll IAP

Ein wesentlicher weiterer Bestandteil der in dieser Arbeit vorgestellten Kommunikations-Infrastruktur ist eine Funktionseinheit, die die Kommunikation zwischen zentral organisierten Recheneinheiten untereinander sowie mit externen Steuerungskomponenten (Antriebe, Aktuatoren und Sensoren) ermöglicht. Zu diesem Zweck wurde am Institut für Elektrische Messtechnik und Grundlagen der Elektrotechnik das **Industrial Automation Protocol (IAP)** spezifiziert [21] [22]. Die Originalspezifikation wurde als Teil der vorliegenden Arbeit im Zuge der schrittweisen Realisierung und Anwendung innerhalb des SFB 562 erweitert und erstmals vollständig implementiert.

Das IAP ordnet sich in die Schicht 3 (Netzwerk) des ISO/OSI-Schichtenmodells ein. Es ist damit die auf dem in Abschnitt 4.4 vorgestellten physikalischen Kommunikationssystem aufsetzende Protokollebene; sie stellt der in Abschnitt 4.2 vorgestellten Middleware die Funktionalitäten zur Kommunikation mit externen Steuerungskomponenten zur Verfügung. In diesem Abschnitt erfolgt zunächst die übersichtsartige Vorstellung des Funktionsumfangs des IAP, dann eine vertiefte Darstellung der zur Realisierung notwendigen Modifikationen und schließlich die Beschreibung der Realisierung im Zusammenhang mit den angrenzenden Softwaremodulen und eine abschließende Leistungsbetrachtung. Dabei wurden Teile der hier dargestellten Sachverhalte

bereits in [132] ausgeführt.

Die Modellierung und Simulation der IAP-Spezifikation sowie modifizierter Funktionalitäten wurde in enger Zusammenarbeit mit dem Institut für Programmierung und Reaktive Systeme (IPS) der TU Braunschweig, dabei insbesondere dem Teilprojekt A5 „Softwaretechnik und formale Analyse“ des SFB 562, durchgeführt. Als grundlegende Vereinbarung für funktionsbeschreibende Diagramme hat sich dabei die Nutzung von UML [1] als besonders praktikabel erwiesen.

### 4.3.1 Zusammenfassung der Funktionsweise

Obwohl das IAP prinzipiell nicht an ein spezielles Kommunikationssystem gebunden ist, wurde es in dieser Arbeit in Anlehnung an die Funktionsspezifikation des IEEE1394-Kommunikationsstandards und dessen effizienten Einsatzes innerhalb der Arbeiten des SFB 562 konzipiert. Und damit für Aufgaben, die ein hohes Maß an Taktsynchronität, Datenquerverkehr und einen hohen Datendurchsatz erfordern. Im weiteren Verlauf der Arbeiten zu diesem Thema wird am Institut für Datentechnik und Kommunikationsnetze der TU Braunschweig in Zukunft eine Realisierung des IAP auch auf Basis eines Industrial-Ethernet-Standards entwickelt [133], um so der IAP-Spezifikation ein breiteres Anwendungsfeld zu eröffnen.

Folgende funktionale Erweiterungen gegenüber dem IEEE1394-Standard werden dabei über die Anwendung des IAP in seiner ursprünglichen Spezifikation definiert:

- Festlegung von Teilnehmerklassen und deren Funktionalitäten
- Aufteilung zyklischer Kanäle für Sollwerte, Istwerte und zyklische Datenquerverkehrbeziehungen
- Festlegung der Interpretation von Dateninhalten
- Nutzung der asynchronen Datenübertragung als Parameterkanal
- Definition eines Netzwerkmanagements
- Nutzung eines Objektverzeichnisses zur Zusammenfassung von Teilnehmerparametern
- Definition von Parametern für einen hochgenauen, synchronen Betrieb der Netzwerkteilnehmer
- Möglichkeiten zur Funktionsüberwachung des Applikationsmasters und der Slaves
- Unterstützung von Teilnehmern mit eingeschränktem Funktionsumfang
- Möglichkeiten zur Begrenzung der Teilnehmerbelastung durch den zyklischen Datenstrom

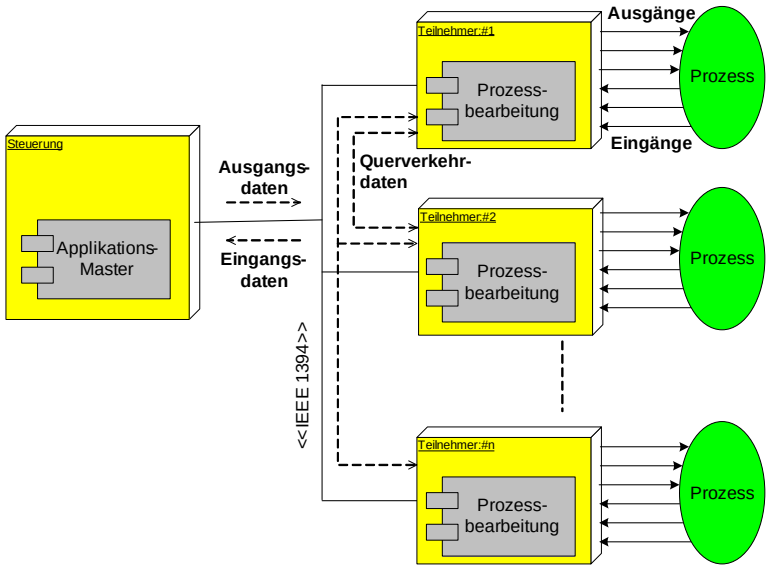



Abbildung 4.31: Netzwerkteilnehmer und ihre Datenbeziehung

Ein Netzwerk im Sinne des IAP besteht aus Teilnehmern mit unterschiedlichen Fähigkeiten. In Abbildung 4.31 ist die physikalische Beziehung der Netzwerkteilnehmer in einem UML-Verteilungsdiagramm dargestellt. Im IAP wird ein **Applikations-Master (AM)** definiert, der als zentrale Steuerung die Vorgaben für die gestellte Applikationsaufgabe an die anderen Teilnehmer vorgibt und zeitlich synchronisiert. Hierfür versorgt der AM die Teilnehmer zyklisch mit neuen Ausgangsdaten, sammelt deren Eingangsdaten ein und wertet diese aus. Die Teilnehmer können zusätzlich über Querverkehrbeziehungen untereinander Daten austauschen und somit Teilaufgaben selbständig und eigenverantwortlich übernehmen. Es erfolgt lediglich eine Statusmeldung und die Synchronisation mit dem AM. Mit Hilfe dieser Beziehungen können z.B. elektrisch gekoppelte Antriebe realisiert oder die Daten schneller digitaler Ein-/Ausgänge auf mehrere Teilnehmer verteilt werden. Wenn die Teilnehmer genügend Rechenleistung besitzen, um neben der Steuerung ihres zugeordneten Prozesses auch übergeordnete Applikationsaufgaben zu übernehmen, und ein Mechanismus zur Verteilung der Anwendung zur Verfügung steht, kann der AM als eigenständige Steuerung auch wegfallen. Dafür ist der zyklische Datenquerverkehr der Teilnehmer Grundvoraussetzung, um die Aufgaben zu koordinieren und Teilprozesse zu synchronisieren. Aber auch der Einsatz mehrerer AM ist denkbar und wird vom IAP unterstützt. Hierbei wird allerdings eine eindeutige Zuordnung der Teilnehmer zu einem AM gefordert, wie das z.B. auch bei der Multi-Master-Anordnung beim Profi-Bus der Fall ist.

Abbildung 4.32 zeigt alle Teilnehmer in einem UML-Klassendiagramm. Dieses Diagramm enthält neben den Beziehungen der Teilnehmer untereinander bereits die im IAP definierten Attribute und Operationen. Für den Transport zyklischer Daten wird auf dem IEEE1394-Bus ein isochroner 125$\mu s$-Zyklus erzeugt, in dem jeder Teilnehmer garantiert ein isochrones Paket senden kann. Die Erzeugung eines aktuellen Pro-

[1]UML = Unified Modeling Language

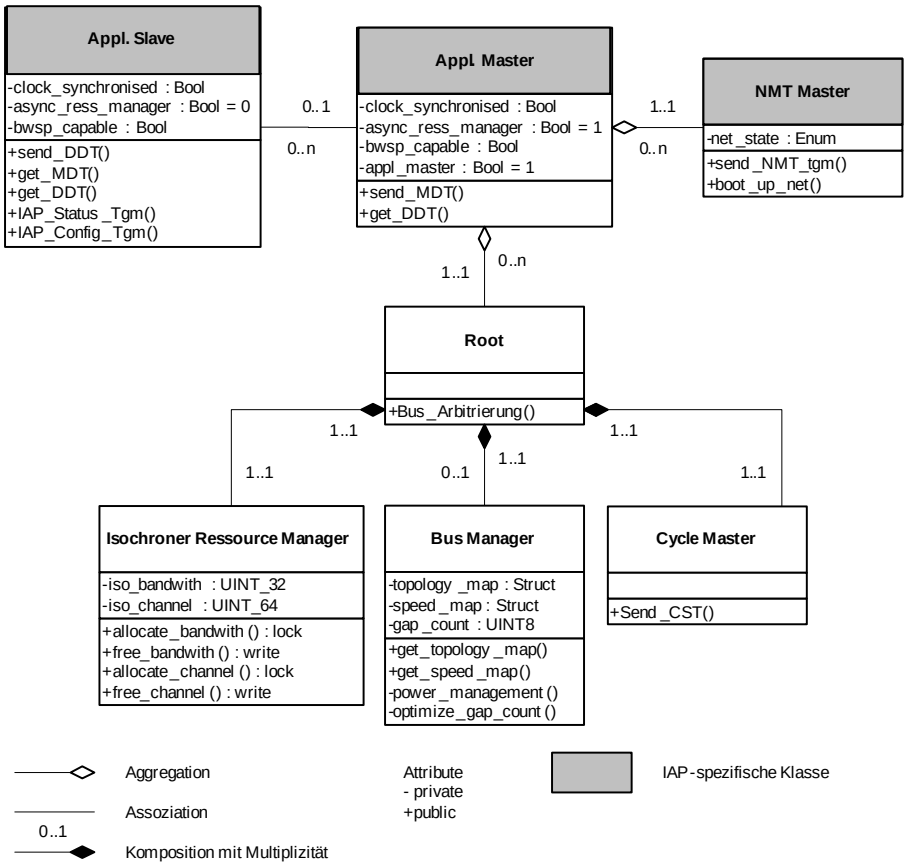



Abbildung 4.32: UML-Klassendiagramm der IAP-Teilnehmer

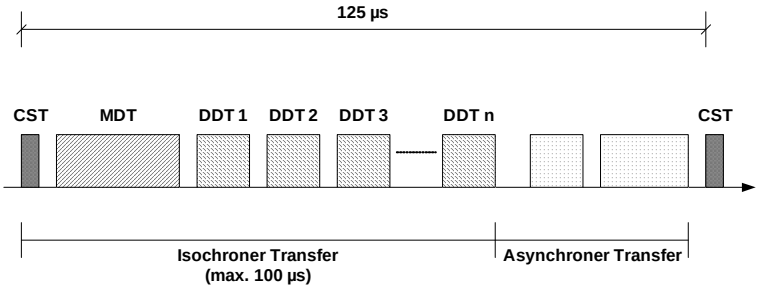



Abbildung 4.33: Übertragungszyklus des IAP

zessabbildes im AM und aktuellen Ausgangsdaten in allen Teilnehmern erfordert die Übertragung aller relevanten Prozessvariablen in jedem Bus-Zyklus. Hierzu müssen gleichzeitig die Eingangsdaten der Teilnehmer zum Master und die Ausgangsdaten vom Master zu den Teilnehmern übertragen werden. Zusätzlich können noch Querverkehrdaten zwischen den Teilnehmern übertragen werden, die ein schnelles Reagieren auf zusammenhängende Ereignisse ermöglichen.

Hierfür definiert das IAP einen Prozess-Datenkanal mit zwei isochronen Paketformen (Abbildung 4.33). Das **Master Data Telegramm (MDT)** wird vom AM zu Beginn eines jeden Zyklus nach dem Cycle Start Telegram (CST) gesendet. Es enthält die Ausgangsdaten für alle Teilnehmer in einem zusammenhängenden Datenstrom. Die folgenden **Device Data Telegramme (DDT)**, von denen jeder Teilnehmer eines pro Zyklus senden darf, enthalten die Informationen, die der jeweilige Teilnehmer zyklisch im Netz bekannt geben will. Jeder andere Teilnehmer kann diese Daten mithören und entscheiden, ob diese für ihn relevant sind und eine Auswertung erfolgen soll oder nicht. Es handelt sich somit um die Umsetzung eines Publish/Subscribe-Mechanismus.

Weiter definiert das IAP einen Parameter-Datenkanal, der über asynchrone Telegramme zur Konfiguration der Teilnehmer und zum Austausch asynchroner Informationen zwischen den Teilnehmern genutzt werden kann. Dieser Datenkanal ermöglicht auch den Zugriff auf alle Parameter und Funktionen jedes Teilnehmers, die dieser dem Netzwerk zur Verfügung stellt. Ein im IAP definiertes Netzwerkmanagement (NMT) ermöglicht die Steuerung der Teilnehmer und damit das definierte Initialisieren des Netzwerks. Dazu wird über den Parameterdatenkanal mit Hilfe von Statustelegrammen eine in jedem Teilnehmer vorhandene Zustandsmaschine gesteuert bzw. abgefragt. Diese Zustandsmaschine, die in Abbildung 4.34 dargestellt ist und deren Phasen in jedem Teilnehmer durchlaufen werden müssen, garantiert das fehlerfreie Initialisieren teilnehmerinterner Strukturen.

Ein Wechsel zwischen den Zuständen erfolgt entweder automatisch durch einen internen Trigger oder wird durch ein NMT-Telegramm vom NMT-Master ausgelöst. Die Übertragung über ein Kommunikationssystem besitzt eine natürliche Übertragungsfehlerwahrscheinlichkeit, die maßgeblich durch die technische Realisierung des Systems und die Umgebungsbedingungen bestimmt wird. Im industriellen Umfeld ist mit starken elektromagnetischen Störungen zu rechnen, die eine Übertragung auf einer Kupferleitung stören können. Zur Erkennung dieser Fehler bietet der IEEE1394-Standard bereits Fehlererkennungsmechanismen durch einen CRC-Code in jedem

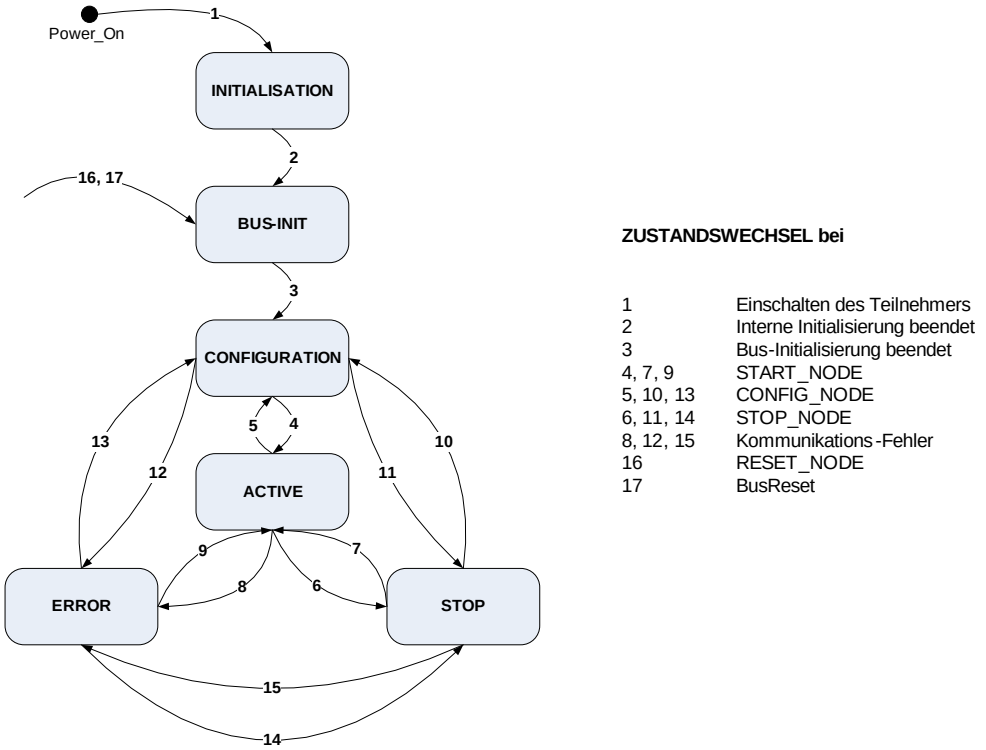



Abbildung 4.34: Zustandsmaschine in den Teilnehmern

asynchron übertragenen Telegramm. Die Fehlerauswertung erfolgt dann im Link Layer Controller und bewirkt ein automatisches Wiederholen der fehlerhaft empfangenen Nachrichten, bevor diese an die Applikationsschicht weitergeleitet werden. Zur Überwachung der isochron übertragenen Daten ist über das IAP in jedem Teilnehmer ein Fehlerzähler vorgesehen, der das Ausbleiben oder fehlerhafte Empfangen von MDT oder DDT registriert und bei Überschreiten einer zulässigen Fehleranzahl den fehlerhaften Teilnehmer in einen Fehlerzustand versetzt und den AM benachrichtigt. Abbildung 4.35 verdeutlicht das Prinzip zur Überwachung isochroner Daten.

Nach jeder korrekt erfolgten isochronen Übertragung wird der Fehlerzähler um eins dekrementiert, damit sich der Teilnehmer von kurzzeitigen, unkritischen Störungen selbständig erholen kann (linker Bildteil). Der rechte Bildteil zeigt das Erreichen des Fehlermaximums in einem Busteilnehmer und den darauf folgenden Wechsel in den Fehlerstatus. Dieser Status kann nach Behebung des Fehlers über ein Konfigurationstelegramm verlassen werden, so dass der Teilnehmer wieder an der isochronen Buskommunikation teilnimmt.

Die Variablen eines jeden Teilnehmers im Netzwerk werden in einer definierten Weise dem Netz zur Verfügung gestellt. Hierfür sieht der IEEE1394-Standard die Verwendung der in der ISO/IEC 13213:1994 beschriebenen Architektur eines Control und Status-Registers (CSR) für Mikrocontroller-Busse vor.

Die Spezifikation des CSR ist in drei Bereiche unterteilt:

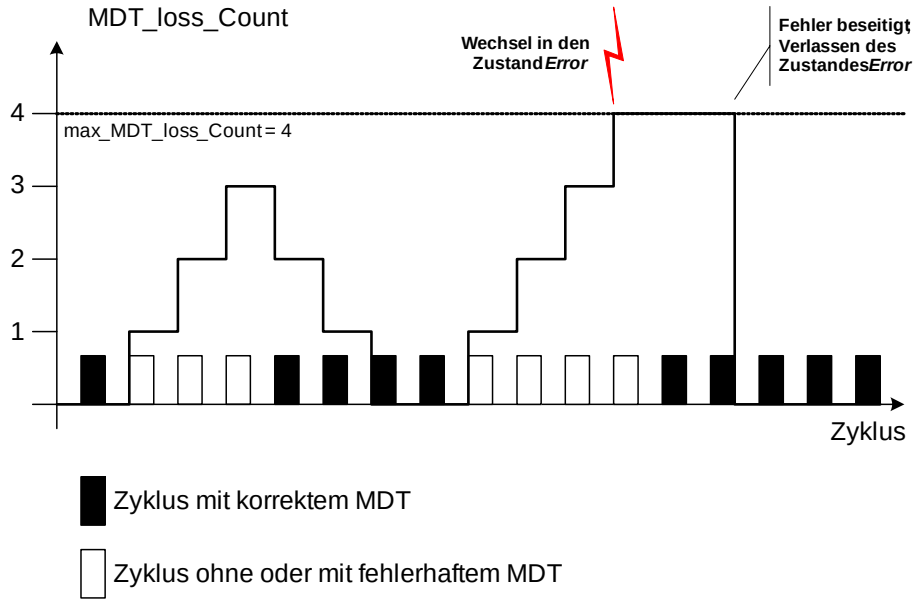



Abbildung 4.35: Behandlung isochroner Fehler

- In ISO/IEC 13213:1994 geforderte Kernregister
- Im IEEE1394-Standard beschriebene busabhängige Register
- Configuration-ROM

Der Speicherbereich des *Configuration-ROM* schließt sich an den Bereich der busabhängigen Register an und enthält eine Verzeichnisstruktur (Root-Directory), die Informationen zur Konfiguration und Diagnose eines Gerätes zur Verfügung stellt. In diese Struktur wird auch das in Anlehnung an den CAN-Standard entwickelte Objektverzeichnis des IAP eingepasst, in welchem die Daten eines jeden Teilnehmers im Netzwerk über unterschiedliche Offsets zugänglich sind.

Eine geschlossene Regelung von Teilnehmern, insbesondere von Antrieben über einen Kommunikationsbus, erfordert die genaue Definition zeitlicher Zusammenhänge innerhalb eines jeden isochronen Übertragungszyklus. Eine solche Definition erfolgt im IAP über Zeitparameter, die global für alle Teilnehmer im Netzwerk gelten. Abbildung 4.36 stellt diese Zusammenhänge dar. Die auf dem Bus übertragenen Telegramme werden zu festgelegten Zeiten, relativ zum Start eines jeden Zyklus, als registriert vorausgesetzt. So ergeben sich feste Zeitpunkte zum Lesen der Eingänge und zum Setzen der Ausgänge in den Teilnehmern, aber auch zum Triggern der Applikation im Master. Istwerterfassung, Übertragung und Sollwertvorgabe sind damit taktsynchron zum Regler.

Neben den hier beschriebenen Eigenschaften des IAP besteht noch die Möglichkeit, Teilnehmer, die nicht in jedem Zyklus ihre isochronen Daten senden können oder brauchen, über spezielle Mechanismen in das Netzwerk zu integrieren. Das Bandbreiten-Splitting teilt dabei die isochronen Kanäle auf mehrere Zyklen auf, wobei das Cycle-

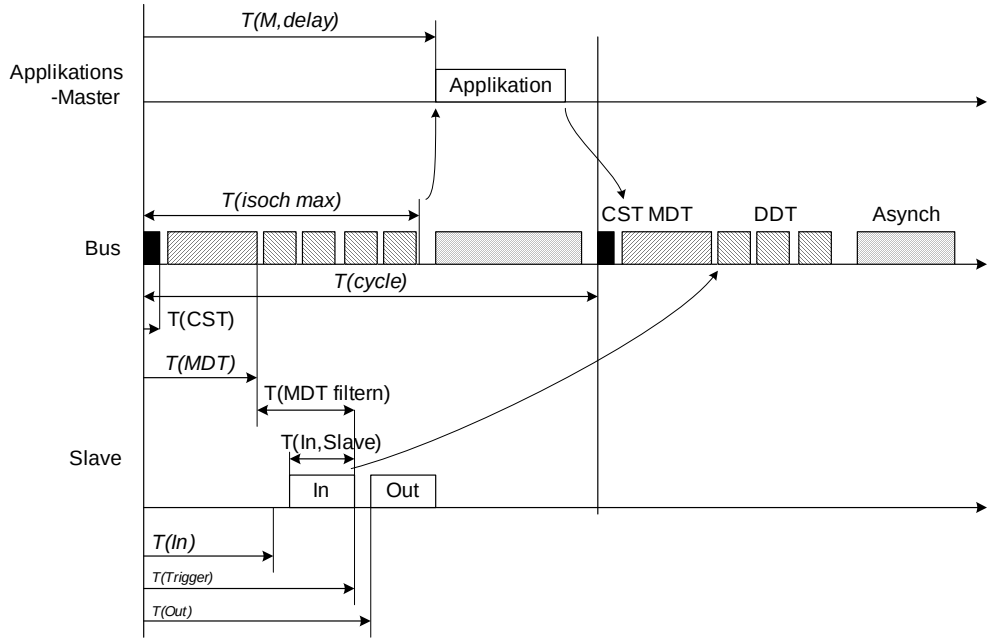



Abbildung 4.36: Zeitparameter des IAP

Skipping lediglich eine bestimmte Anzahl von Zyklen abwartet, bevor eine isochrone Übertragung des Teilnehmers erfolgt.

### 4.3.2 Modifikation und Erweiterung des IAP

In der ursprünglichen Spezifikation des IAP erfolgte das Übertragen des MDT und der DDT direkt zu Beginn eines jeden Regelungszyklus, eingeleitet durch das Erscheinen eines **Cycle Start Telegrams (CST)** zu Beginn des isochronen Transfers. Abbildung 4.36 zeigt den zugehörigen zeitlichen IAP-Ablauf für einen solchen Regelungszyklus. Dieser spezifikationsgemäße Zyklusablauf weist allerdings wesentliche Probleme hinsichtlich der Anwendbarkeit innerhalb einer komplexen Steuerung auf:

**Begrenzte Dynamik der Regelung** Die ursprüngliche IAP-Spezifikation sieht vor, dass bereits zu Beginn eines jeden Kommunikationszyklus alle Soll- und Istwerte für den Kommunikationsvorgang zwischen der zentralen Steuerung und den externen Steuerungskomponenten feststehen. Diese Festlegung entspricht auch der IEEE1394-Spezifikation für isochrone Telegramme. Auf diese Weise können alle innerhalb der Regelung benötigten Telegramme in einem relativ kurzen Zeitfenster zu Beginn des Regelungszyklus übertragen werden. Allerdings besteht zwischen den in den Telegrammen übertragenen Soll- und Istwerten des gleichen Kommunikationszyklus eine effektive Zeitverzögerung, die mindestens der Größe einer Zykluszeit entspricht. Die Istwerte entsprechen also nicht mehr dem aktuellen, sondern dem vergangenen Regelungszyklus. Diese Zeitverzögerung begrenzt damit die Dynamik der Regelung und muss dort rechnerisch berücksichtigt werden.

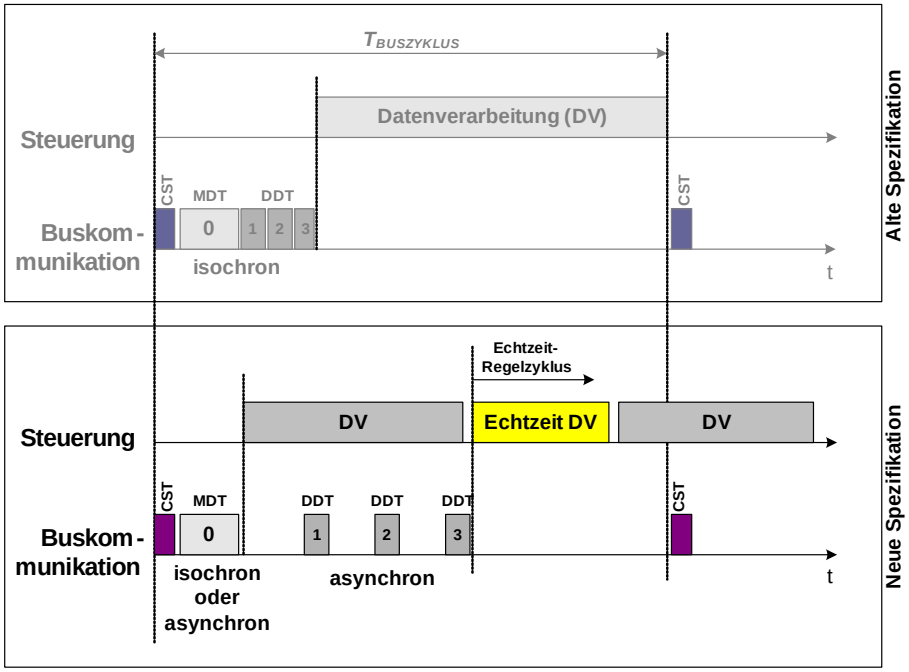



Abbildung 4.37: Modifizierter IAP-Zyklus

**Problematische zeitliche Zuordnung** Durch den Umstand, dass die Istwerte (DDT) aus Sicht der externen Steuerungskomponenten nicht sofort, sondern immer erst frühestens zu Beginn des nächst folgenden Kommunikationszyklus versandt werden, kann kein Bezug mehr zum tatsächlichen Alter dieser Daten hergestellt werden bzw. die Auflösung der Zeitzuordnung liegt bei einer Zykluszeit. Je schneller also eine Steuerungskomponente ihre Istwerte zur Verfügung stellt, umso länger warten diese Daten effektiv bis zur tatsächlichen Übertragung auf dem Bus. Dies schränkt die erreichbare Regelgüte weiter ein.

Im Zuge der Realisierung und Inbetriebnahme des IAP am Roboter erfolgten Funktionsmodifikationen im Bereich der verwendeten Datenstrukturen, des Kommunikationsablaufs und der Verwendung von IEEE1394-spezifischen Telegrammtypen. Abbildung 4.37 stellt den modifizierten Kommunikationsablauf im unteren Teil dar, der zur Realisierung und Inbetriebnahme der SFB-Roboter verwendet wurde. Die Modifikationen werden im Folgenden aufgeführt.

### Datenstrukturen

In der ursprünglichen IAP-Spezifikation wurde für die externen Steuerungskomponenten der Aufbau je eines individuellen DDT und eines universellen MDT definiert. Die Adressierung sah dabei vor, dass jede Komponente mehrere (bis zu $2^{16}$) unterschiedliche Variablen verarbeiten könnte. In der derzeitigen Realisierung wird davon ausgegangen, dass nur eine einzige Applikation innerhalb der jeweiligen Steuerungskomponente zur Verarbeitung der Soll- und Istwerte verwendet wird. Damit reicht

auch die Übertragung *einer* Variablen, die allerdings strukturiert sein kann. Weiter wurde der Bandbreitenbedarf für die Übertragung von Telegrammen reduziert, indem anstatt der Verwendung von der IEEE1394-Spezifikation entsprechenden 48-Bit-Adressen nun nur 32-Bit-Adressen verwendet werden, welche einen Arbeitsspeicherbereich abdecken, der noch immer viel größer ist als der Bedarf.

### Verwendung der Telegrammtypen

Anders als in der ursprünglichen Spezifikation werden für die Übertragung der DDT keine isochronen, sondern (bedarfsorientierte) asynchrone Telegramme verwendet. Durch diese Modifikation besteht die Möglichkeit, Sollwerte (MDT) nun nicht sofort, sondern erst dann mit den Istwerten (DDT) zu beantworten, wenn innerhalb der jeweiligen Steuerungskomponente eine den Sollwerten entsprechende Reaktion der Regelstrecke erfolgt und im Optimalfall sogar abgeschlossen ist. Der Zeitpunkt, wann ein DDT auf dem Bus übertragen wird, liegt damit möglichst nahe an der tatsächlichen Messung, so dass die zentrale Steuerung nun aktuelle Prozessdaten vorfindet und zur Generierung neuer Sollwerte verarbeiten kann. MDT können in der derzeitigen Realisierung wahlweise mit isochronen oder asynchronen Telegrammen übertragen werden. Im ersten Fall lässt sich eine große Zyklussteifheit im $125\mu s$-Raster erreichen, während der zweite Fall die Entwicklung eines zukünftigen Konzepts zur Erhöhung des (im Allgemeinen) festen, maximalen $125\mu s$ Buszyklus unterstützt, das in Abschnitt 6 erläutert wird. CST werden nur von der zentralen Steuerung ausgewertet, um innerhalb der externen Steuerungskomponenten Verarbeitungszeit-Ressourcen zu sparen. Diese reagieren ausschließlich auf den Empfang oder das Ausbleiben von MDT.

### Timing

Die ursprüngliche IAP-Spezifikation sieht vor, dass für alle externen Steuerungskomponenten sowie die zentrale Steuerung eine einheitliche Zeitbasis mitsamt globalen Zeitparametern auf den Empfangszeitpunkt des CST basieren. In der aktuellen Realisierung wird das CST nur verwendet, um eine lokale Zeitreferenz für die zentrale Steuerung zu erzeugen. Dort wird daraufhin das MDT zusammengestellt und abgeschickt. Jede Steuerungskomponente empfängt dieses MDT und extrahiert lokal relevante Sollwerte, die die jeweiligen Stellgrößen für einen Regelstreckenteil erzeugen. Nach einem individuell festlegbaren Zeitverzug werden in den entsprechenden Steuerungskomponenten lokal die Sensordaten eingelesen und als DDT auf den Bus geschrieben. Sobald die zentrale Steuerung das letzte erwartete DDT eingelesen hat, beginnt direkt die Echtzeit-Auswertung der Sensordaten in entsprechenden Regelungsmodulen und das erneute Zusammenstellen der MDT für den kommenden Regelungszyklus.

Über dieses Verfahren wird der sequenzielle Betrieb zweier unterschiedlicher Steuerungsebenen ermöglicht (s. Abbildung 4.37): Eine Steuerungsebene, die Funktionsmodule für die harte Echtzeitverarbeitung zur Ausführung bringt und eine andere, innerhalb derer übrige Funktionsmodule prioritätsbasiert zur Ausführung kommen. Weiter können so auch mögliche applikationsbedingte Modifikationen der Zykluszeit

realisiert werden, um damit auf stark variierende Bandbreitenanforderungen reagieren zu können. Beispielsweise, wenn in einem Arbeitspunkt des Roboters eine Bildverarbeitung durchgeführt werden muss, während derer die Struktur steht oder sich nur sehr langsam bewegt und es weniger auf Zykluszeit, sondern eher auf Übertragungsbandbreite ankommt.

Das „Bandbreiten-Splitting“ sowie das „Cycle-Skipping“ der ursprünglichen IAP-Spezifikation werden somit über diesen leistungsfähigeren Mechanismus ersetzt, der aber voraussetzt, dass *jede* Komponente pro Kommunikationszyklus genau ein DDT versenden kann. Effektiv ergibt sich mit der Anwendung dieses Mechanismus die Umstellung des globalen Bus-Synchronisationsmechanismus von zeit- auf ereignisbasiert. Zeitbasiert sind nur noch die Vorgänge *innerhalb* einer jeden Steuerungskomponente, um eine lokale Ablaufsteuerung zu realisieren. Diese applikationsspezifischen Zeitparameter sind allerdings nicht mehr Zuständigkeitsbereich des IAP, sondern müssen von dem entsprechenden Anwender selbst festgelegt und umgesetzt werden. Hierzu existiert allerdings in der Realisierung des IAP ein Funktionsmodul, welches die lokale Umsetzung unterstützt.

Mit der Umstellung von zeit- auf ereignisgesteuert entfällt zunächst auch der Vorteil des systemweit einheitlichen Einlesezeitpunkts von Sensordaten in den externen Steuerungskomponenten. So wäre es theoretisch möglich, dass die Datensynchronisation auf Applikationsebene undurchsichtig bliebe. Diesem Umstand wird allerdings über die Möglichkeit eines systemweit einheitlichen Zeitstempels Rechnung getragen, der direkt über Register des unterlagerten IEEE1394-Kommunikationssystems zugreifbar ist. Bei kritischen Istwerten wäre dieser Zeitstempel in die entsprechende Datenstruktur des jeweiligen DDT einzufügen und innerhalb der Regelungsfunktionen der zentralen Steuerung zu berücksichtigen.

Zu dem Leistungsumfang der hier vorliegenden Arbeit gehört die vollständige Realisierung, Implementierung und Erprobung des IAP zur Anwendung als physikalische und softwaretechnische Kommunikationsgrundlage zur Entwicklung von Parallelrobotern des SFB 562. Sämtliche entwickelte Soft- und Hardware wurde mittels der UML vollständig und einheitlich beschrieben. In Abschnitt 4.3.3 werden grundlegende Funktionalitäten anhand ausgewählter Diagramme genauer beschrieben.

### 4.3.3 Realisierung IAP

Um die Realisierung des IAP zu beschreiben, wird in diesem Absatz die Threadstruktur vorgestellt und in den folgenden Absätzen auf die Konfigurierung und den Start des IAP, die Benutzerschnittstelle zum Zugriff auf das IAP, die Initialisierung des Netzwerks und die Konfigurierung der Knoten durch das IAP sowie die Aktivitäten des IAP innerhalb der aktiven Phase eingegangen.

Abbildung 4.38 zeigt die Threadstruktur, die zum Betrieb des IAP verwendet wird. Insgesamt sind mindestens drei weitere Prozesse (MiRPA-X, IEEE1394 und CONTROL) während des Ablaufs des IAP aktiv, die allesamt mit dem IAP-Prozess in Verbindung treten. Der **MiRPA-X-Prozess** stellt die lokale Kommunikations-

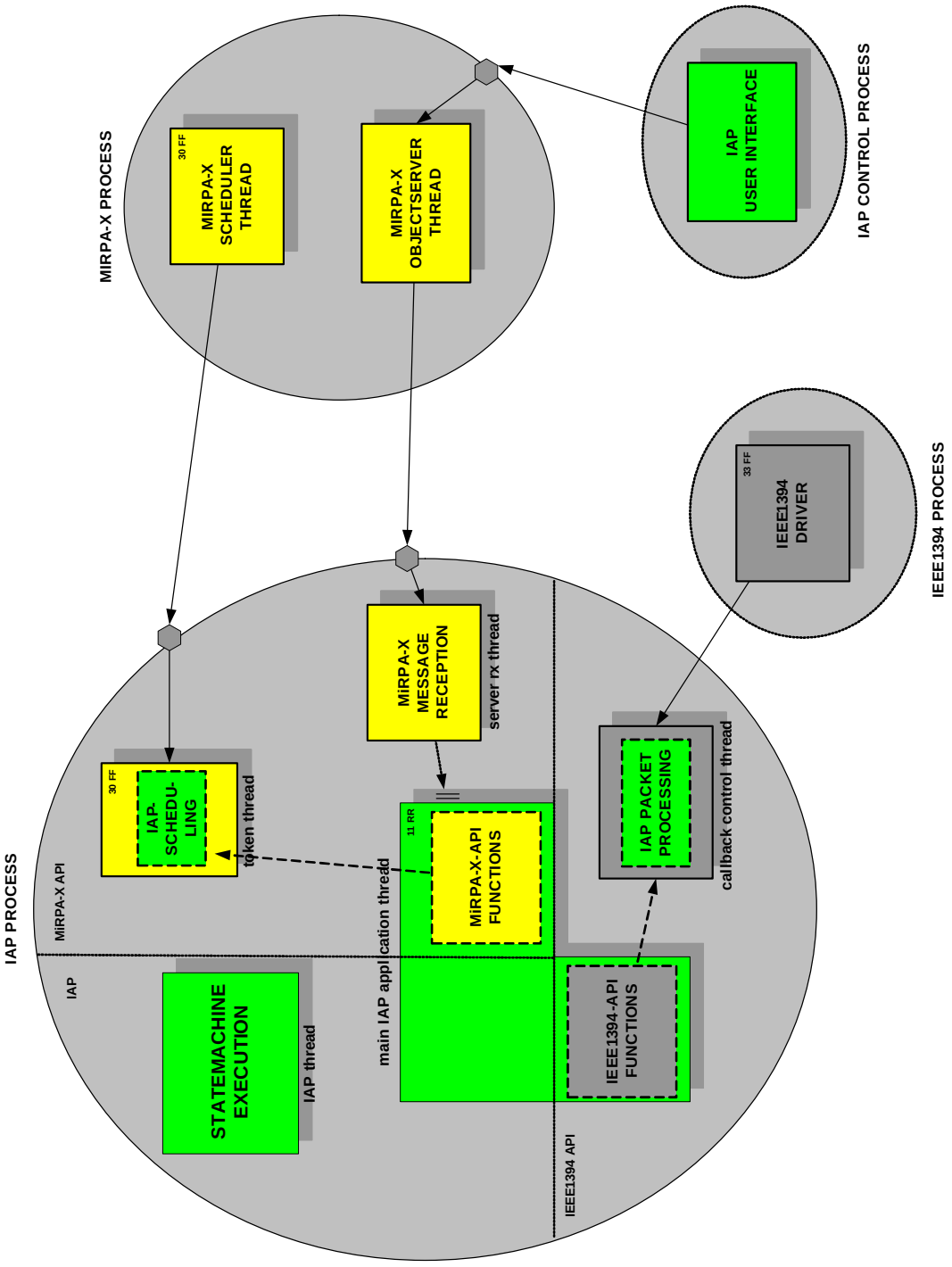



Abbildung 4.38: Threadstruktur des IAP

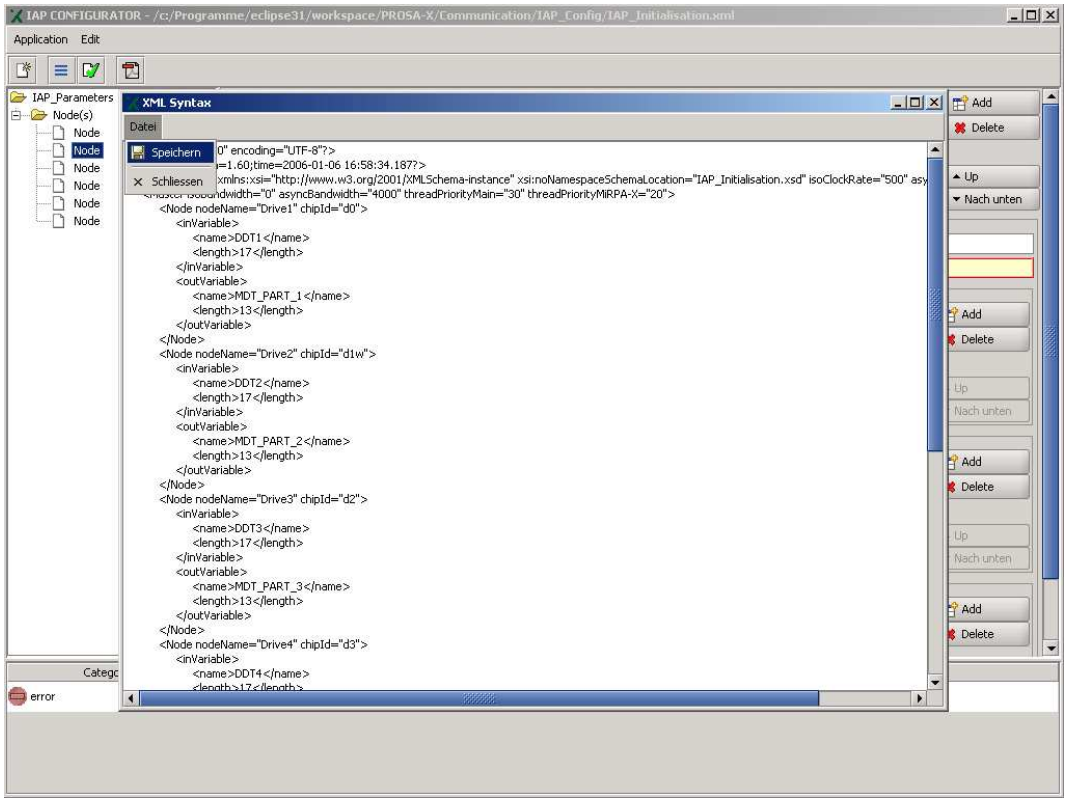



Abbildung 4.39: Screenshot des IapConfigurators

schnittstelle dar und koordiniert das Versenden des Echtzeit-Tokens; über den **IAP-CONTROL-Prozess** kann das Verhalten des IAP während der Laufzeit beeinflusst werden; der **IEEE1394-Prozess** wird in Abschnitt 4.4 vorgestellt und ist die Verbindung zwischen der IAP-Software und den über das externe Kommunikationsmedium zugreifbaren Daten; der **IAP-Prozess** selbst beinhaltet alle im Folgenden vorgestellten Funktionalitäten. Er nutzt dabei die Programmierschnittstellen der Module MiRPA-X und IEEE1394.

### Konfigurierung und Start

Die Konfigurierung des IAP erfolgt über das Einlesen und Auswerten einer XML[2]-Datei, die zuvor vom Anwender erstellt werden muss. Der Dateiname wird beim Start der IAP-Applikation als Kommandozeilenparameter mit angegeben. Da die Konfigurierung mittels textbasierter Dateien u.U. unübersichtlich und strukturell fehleranfällig ist, wurde eine Java-Applikation **IAP-Configurator** entwickelt, die alle benötigten Eingaben über komfortabel nutzbare Eingabemasken erfragt und diese dann automatisch in das XML-Ausgabeformat exportiert.

Abbildung 4.39 zeigt einen Screenshot dieses Tools mit der exportierten XML-Datei im Vordergrund. Innerhalb der XML-Datei werden Angaben gemacht, die die be-

[2]eXtensible Markup Language

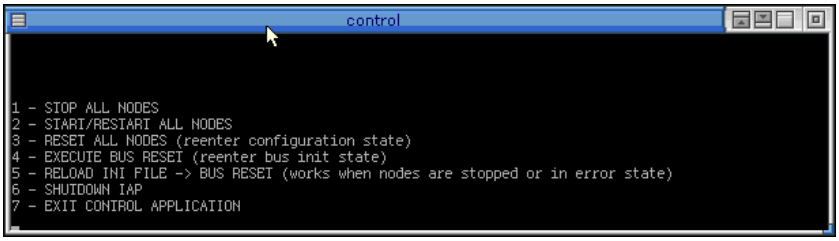



Abbildung 4.40: Screenshot des IAP-Control-Prozesses

absichtigte Kommunikations-Zykluszeit, die Struktur des externen Kommunikationsnetzwerks und die Bandbreite für ein- und ausgehende Daten jedes Kommunikationsteilnehmers betreffen. Auf diese Weise wird auch festgelegt, welcher externe Kommunikationsteilnehmer welchem Shared-Memory-Bereich zugeordnet ist, um seine Daten dort abzulegen bzw. diese von dort zu beziehen. Im späteren Applikationsverlauf sind für alle an MiRPA-X angeschlossenen Prozesse nur noch die Shared-Memory-Bereiche (Identifikation erfolgt über eindeutige Namensvergabe) zugreifbar, in denen die Ist- und Sollwerte der externen Teilnehmer abgebildet und veränderbar sind.

#### Benutzer- und Programmierschnittstelle

Anders als für die Anwendung von MiRPA-X direkt, existiert für die IAP-Master-Applikation keine Programmierschnittstelle. Da der IAP-Master bereits selbst eine Server-Applikation im Sinne von MiRPA-X ist, verwendet sie zur Realisierung intern die im Abschnitt 4.2.2 vorgestellten Funktionsaufrufe und Threads. Zur Nutzung bzw. Anpassung des IAP-Masters an spezielle Wünsche wurde dagegen eine auf MiRPA-X-Nachrichten basierende Benutzerschnittstelle definiert, deren Anwendung exemplarisch über den Prozess „IAP CONTROL“ umgesetzt wurde. Dieser Prozess bietet eine textbasierte Bildschirmschnittstelle, mit deren Hilfe sich Funktionen auswählen lassen, die in Form von Nachrichten an den ObjectServer gesendet werden. Abbildung 4.40 zeigt einen Screenshot des Kontrollprozesses. Hierüber lassen sich eingebaute Funktionalitäten des IAP zur Laufzeit aufrufen, um beispielsweise alle externen Knoten in einen bestimmten Zustand zu versetzen oder die Konfigurationsdatei neu zu laden und die Software auf den externen Knoten neu zu starten. Insgesamt sind folgende COMMAND-Nachrichten definiert, die von beliebigen über MiRPA-X verbundene Clients aufgerufen werden können:

**„IAP: STOP NODE“** Anhalten aller Knoten. Dieses Kommando versetzt alle Steuerungskomponenten in den Zustand STOP. Je nach Vorzustand wird damit die Transition 6, 11 oder 14 durchgeführt (s. Abbildung 4.34).

**„IAP: START NODE“** Starten aller Knoten. Dieses Kommando führt zu einer Transition 4, 7 oder 9 in allen Steuerungskomponenten (s. Abbildung 4.34).

**„IAP: RESET NODE“** Zurücksetzen eines Knotens. Dieses Kommando führt zu der Transition 16 in einer auswählbaren Steuerungskomponente, die beginnend mit dem Zustand BUS_INIT alle Schritte bis zum aktiven Zustand durchläuft (s. Abbildung 4.34).

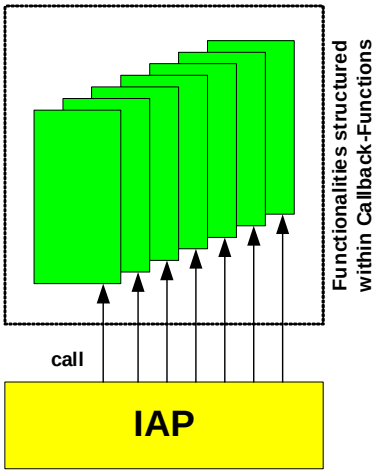



Abbildung 4.41: Schema des Funktionalitätsaufrufs innerhalb der Knoten

**„IAP: SHUTDOWN“** Beenden des IAP. Dieses Kommando beendet die lokale IAP-Master-Applikation und versetzt dabei die externen Steuerungskomponenten in den Zustand STOP.

**„IAP: BUS RESET“** Setzen des Bus-Reset-Signals auf den Kommunikationsleitungen. Dieses Kommando bewirkt das Zurücksetzen sämtlicher Kommunikationshardware und eine Neukonfiguration. Anschließend wird in allen externen Steuerungskomponenten automatisch die Transition 17 durchgeführt (s. Abbildung 4.34).

**„IAP: INI FILE RELOAD“** Dieses lokale Kommando führt dazu, dass die Konfigurationsdatei neu geladen wird und damit das Netzwerk neu konfiguriert wird.

Für die externen Steuerungskomponenten (Knoten) wird dagegen eine Programmierschnittstelle verwendet. Sie basiert auf der Nutzung eines Digitalen Signalprozessors (DSP) und einer zugehörigen C-Entwicklungsumgebung „CodeComposer“ von Texas Instruments. Um die Programmierung der in den externen Steuerungskomponenten ablaufenden Applikationen zu vereinfachen und gleichzeitig sicherzustellen, dass Zyklusaktivitäten nicht durch Programmierfehler innerhalb der Applikation gestört werden können, wird dabei ein Callback-orientierter Ansatz verwendet. Abbildung 4.41 zeigt den Aufruf der Applikationsfunktionalitäten zur Verdeutlichung der Programmierschnittstelle. Über vom IAP fest vorgegebene Funktionsprototypen kann der Anwender einzelne Funktionsmodule gekapselt entwickeln. Dabei werden folgende Funktionen (UserFunction) unterschieden:

**IAP_Node_InitFunction()** beinhaltet den applikationsspezifischen C-Code, der ausgeführt wird, um beispielsweise zusätzlich benötigte Hardware zu initialisieren oder Speicher für die Applikation zu reservieren.

**IAP_Node_BusInitFunction()** beinhaltet C-Code, der im Anschluss an die Bus-Initialisierung ausgeführt wird.

**IAP_Node_ConfigFunction()** beinhaltet C-Code, der bereits auf netzwerkweit verfügbare Informationen, wie z.B. die eingestellte Zykluszeit, zugreifen kann. Hier werden sinnvollerweise auch die CycleTimer-Einstellungen vorgenommen, mit denen zyklusinterne Ereignisse zeitgenau vorgegeben werden können.

**IAP_Node_StopFunction()** beinhaltet applikationsspezifischen C-Code, der ausgeführt werden soll, sobald die Kommunikation mit dem IAP-Master durch ein STOP-Kommando unterbrochen wurde. Hier kann Hardware in einen sicheren Zustand versetzt werden.

**IAP_Node_ErrorFunction()** beinhaltet applikationsspezifischen C-Code, der ausgeführt werden soll, sobald ein Fehlerzustand erreicht ist.

**IAP_Node_ExitFunction()** beinhaltet applikationsspezifischen C-Code, der zur Ausführung kommt, wenn die lokale IAP-Zustandsmaschine in den EXIT-Zustand eintritt. Der Knoten sollte vollständig abgeschaltet werden.

**IAP_Node_Pre_ASYNC_RX_Function()** wird ausgeführt, sobald ein asynchrones Telegramm vom Knoten empfangen wird. In der Regel können damit Zeitgeber zurückgesetzt oder neu gestartet werden, um zyklussynchrone Ereignisse vorzubereiten.

**IAP_Node_MDT_RX_Function()** wird ausgeführt, wenn das IAP ein MDT erkannt hat und die Daten daraus für die Applikation zugreifbar sind.

**IAP_Node_CST_RX_Function()** wird ausgeführt, sobald lokal ein CST empfangen wird. Da die Leistungsfähigkeit der externen Recheneinheiten bislang noch nicht ausreichend ist, um neben der applikationsspezifischen Tätigkeit noch Impulse im 125$\mu s$-Takt zu verarbeiten, wird von einer Nutzung dieser Funktion abgeraten.

Alle diese Funktionen werden an ausgezeichneten Stellen innerhalb der IAP-Zustandsmaschine des Knotens ausgeführt. Und zwar immer *nachdem* die IAP-eigenen Funktionalitäten des entsprechenden Zustands abgearbeitet sind. Beispielsweise kann so innerhalb der *IAP_Node_MDT_RX_Function()* die Auswertung der innerhalb des MDT empfangenen Sollwerte erfolgen und daraufhin weitere externe Hardware angesprochen werden.

Eine detailliertere Beschreibung der Programmierschnittstelle findet sich im Benutzerhandbuch des IAP [134].

### Konfiguration der Knoten und des Netzwerks

Nach dem Einschalten einer externen Steuerungskomponente (Knoten, Teilnehmer) erfolgt im Zustand INITIALISATION das Einrichten der CSR-Strukturen und das Reservieren lokal benötigter Speicherbereiche zum Senden und Empfangen von Telegrammen über den Bus. Der daran anschließende Zustand BUS-INITIALISATION dient dazu, die Kommunikationshardware auf den Empfang von Telegrammen vorzubereiten und Schnittstellen-Callbacks festzulegen.

Innerhalb des Zustands CONFIGURATION erfolgt nun die Identifikation und Konfigurierung der Knoten. Dabei erfolgt das Versenden einer Statusabfrage vom Master an alle Knoten (NMT-Kommando), die von allen angeschlossenen Knoten beantwortet wird. Ist die ID des jeweiligen Knotens innerhalb des Masters bekannt, so wird dieser als im Zustand CONFIG befindlich registriert und seine Bandbreitenanforderung aus der XML-Vorgabe umgesetzt. Nun versendet der Master Konfigurationstelegramme als Broadcast, die neben einer eindeutigen Ziel-Knoten-ID auch die jeweiligen Vorgaben für Bandbreite und Datenstruktur übertragen. Sind alle Konfigurationstelegramme übertragen, so schließt ein PrepareActivation-Telegram diesen Teil der Konfiguration ab und der Master wartet auf das Eintreffen entsprechender Statustelegramme aller konfigurierten Knoten. Daraufhin versetzt der Master die Knoten in den Zustand ACTIVATE, der durch das Versenden eines StartNode-Kommandos ausgelöst wird.

### Aktivitäten während des Betriebs

Abbildung 4.42 zeigt das UML-Aktivitätsdiagramm, das die Vorgänge innerhalb des IAP-Masters für den Zustand ACTIVE beschreibt. In diesem Zustand ist der Nutzdatentransfer der höheren Softwareebenen aktiv, so dass es den Funktionalitäten der Steuerung ermöglicht ist, netzwerkweit Daten auszutauschen. Alle Aktivitäten des IAP werden Master-seitig über den Telegrammempfang aus der unterlagerten Kommunikationsebene gestartet, welcher zunächst in den Aufruf einer entsprechenden Callback-Funktion mündet. Handelt es sich um ein im Kommunikationszyklustakt einkommendes CST (innerhalb der unterlagerten IEEE1394-Kommunikationsschicht aus den $125\mu s$-Impulsen abgeleitet), so wird überprüft, ob seit dem letzten Versenden eines MDT alle erwarteten DDT der Knoten eingetroffen sind. Ist das der Fall, so wird der Echtzeit-Token (s. Abschnitt 4.2) angefordert, der erst wieder nach dem Empfang des letzten DDT freigegeben wird und das bereits neu berechnete MDT über den Bus an die Knoten verschickt.

Beim Eintreffen eines DDT wird zunächst der Sender-Knoten bestimmt und die Telegrammdaten anhand der in der XML-Konfigurationsdatei vereinbarten Zuordnung in die dafür vorgesehenen Shared-Memory-Bereiche geschrieben. Zum Zwecke einer Geschwindigkeitsoptimierung wird die Nutzung eines Blocktransfers empfohlen, der nicht das Eintreffen jedes DDT, sondern aller DDT zusammengefasst signalisiert und die Daten in einem Block vorhält. Nach der Auswertung jedes DDT erhöht sich der Zähler für bereits empfangene DDT um eins. Sobald alle erwarteten DDT empfangen wurden (die Daten befinden sich dann in entsprechenden Shared-Memory-Bereichen), bewirkt das Freigeben des Echtzeit-Tokens, dass nun alle weiteren Echtzeitprozesse im System zur Anwendung kommen und auf die Nutzdaten zugreifen.

Genau wie auch der IAP-Master, verhalten sich die Knoten im aktiven Zustand ereignisgesteuert. Dabei wären auch Ereignisse denkbar, die dem lokalen Applikationsprozess entspringen und im Fehlerfall zur definierten Unterbrechung des aktiven Kommunikationsvorgangs führen können. An dieser Stelle sind allerdings nur die das Kommunikationssystem betreffenden Ereignisse von Interesse. Abbildung 4.43 zeigt das UML-Aktivitätsdiagramm für den Zustand ACTIVE innerhalb jedes Knotens. Ein

von der IEEE1394-Kommunikationsschicht empfangener Interrupt führt zur Ausführung einer entsprechenden Callback-Funktion, die beim Eintreffen eines MDT dieses auswertet und die für den lokalen Knoten relevanten Daten für die zyklischen Anwenderfunktionalitäten zugänglich macht. 147

Während innerhalb des IAP-Masters das Versenden des MDT sowie der Empfang der DDT automatisch geschieht, muss das Versenden des DDT innerhalb der Knoten vom Anwender initiiert werden. Nur er weiß, wann beispielsweise die aktuellen Sensordaten gültig bzw. Berechnungen abgeschlossen sind. Daher ist das Versenden der DDT nicht in der Abbildung 4.43 dargestellt; es erfolgt zu einem applikationsspezifischen Zeitpunkt. Erfolgt aber das Versenden des DDT so spät, dass es nicht mehr innerhalb des aktuellen Regelzyklus verarbeitet werden kann, so führt dies zu einem Ablauffehler, welcher alle an der IAP-Kommunikation beteiligten Komponenten in den Zustand ERROR versetzt, bis dieser Fehlerzustand behoben ist.

### Typische Ablaufsequenz des IAP

Spezifikationsgemäß finden die Aktivitäten des IAP-Netzwerkmanagements, bei dem große Datenmengen übertragen werden müssen, nicht während der echtzeitkritischen, sondern der Konfigurationsphase des IAP statt. Die kritische Echtzeitkommunikation (Zustand AKTIV) ist im IAP so realisiert, dass sich die Kommunikationsvorgänge auf das Übermitteln aktueller Datenzeiger an die angrenzenden Applikationen (Steuerung und Kommunikationshardware) beschränken. Abbildung 4.44 zeigt die prinzipiell notwendigen IAP-Aktivitäten zur Verdeutlichung der Zeitparameter bei den wesentlichen zyklischen Kommunikationsvorgängen in einem Sequenzdiagramm.

Der Start eines neuen Regelungszyklus ist eng mit dem Start eines Kommunikationszyklus verknüpft. Über den Empfang des CST wird innerhalb des IEEE1394-Treibers PC-seitig ein daraus abgeleiteter Regelungs-Zyklusstart für die Steuerungsapplikation erzeugt, für den die neuen Sollwerte an die externen Knoten verteilt werden sollen. Sobald vom IEEE1394-Treiber also ein CST empfangen ist, wird daraufhin innerhalb der Zeitspanne `T_CST_QNX` eine Callback-Funktion ausgeführt, in der die Prüfung erfolgt, ob bereits alle erwarteten DDT empfangen worden sind. Ist das der Fall, so wird weiter geprüft, ob der Echtzeittoken von MiRPA-X bereits wieder an das IAP versendet worden ist (als Zeichen dafür, dass auch alle Token-Thread der Applikation erfolgreich abgelaufen sind). Nun erfolgt die MDT-Zusammenstellung und die Beauftragung des IEEE1394-Treibers, das vorliegende MDT auf den Bus zu schreiben. Die Zeit, die für das Verschicken des Telegramms verstreicht, hängt dabei stark von der Realisierung der Funktionalität innerhalb des IEEE1394-Treibers ab. Nach Versand des Telegramms kann innerhalb des IAP während der Zeit `T_Tx_QNX` geprüft werden, ob es zu einem Kommunikationsfehler gekommen ist. Diese Funktionalität des Treibers wird allerdings nicht genutzt, da innerhalb des IAP andere Fehlererkennungsmechanismen verwendet werden. Anschließend können andere Applikationsprozesse ihre Arbeit weiterführen.

Innerhalb der Knoten (DSP) werden die Telegramme über den IEEE1394-Treiber empfangen, sobald sie auf dem Bus liegen. Wenn die Daten vollständig im RAM

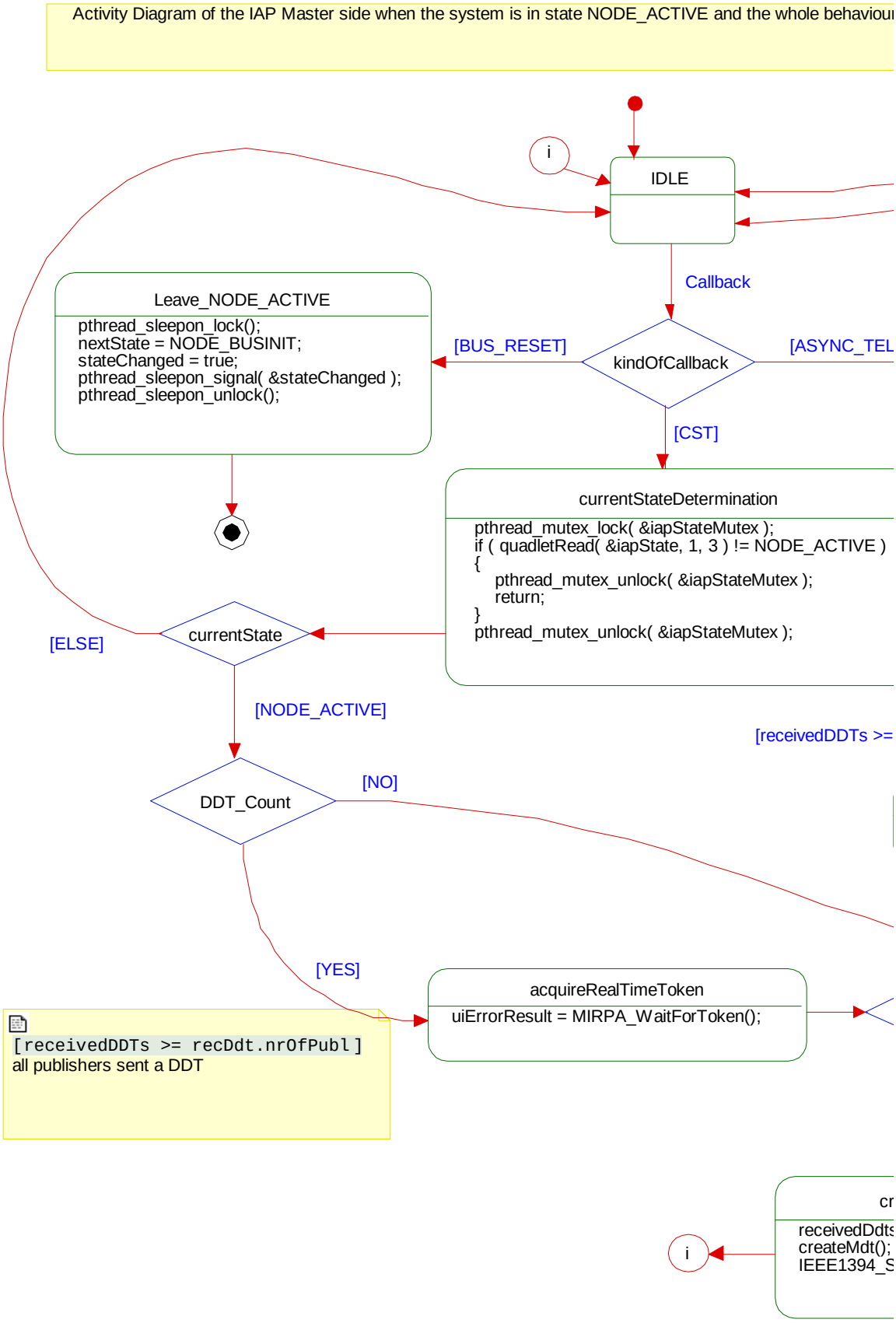
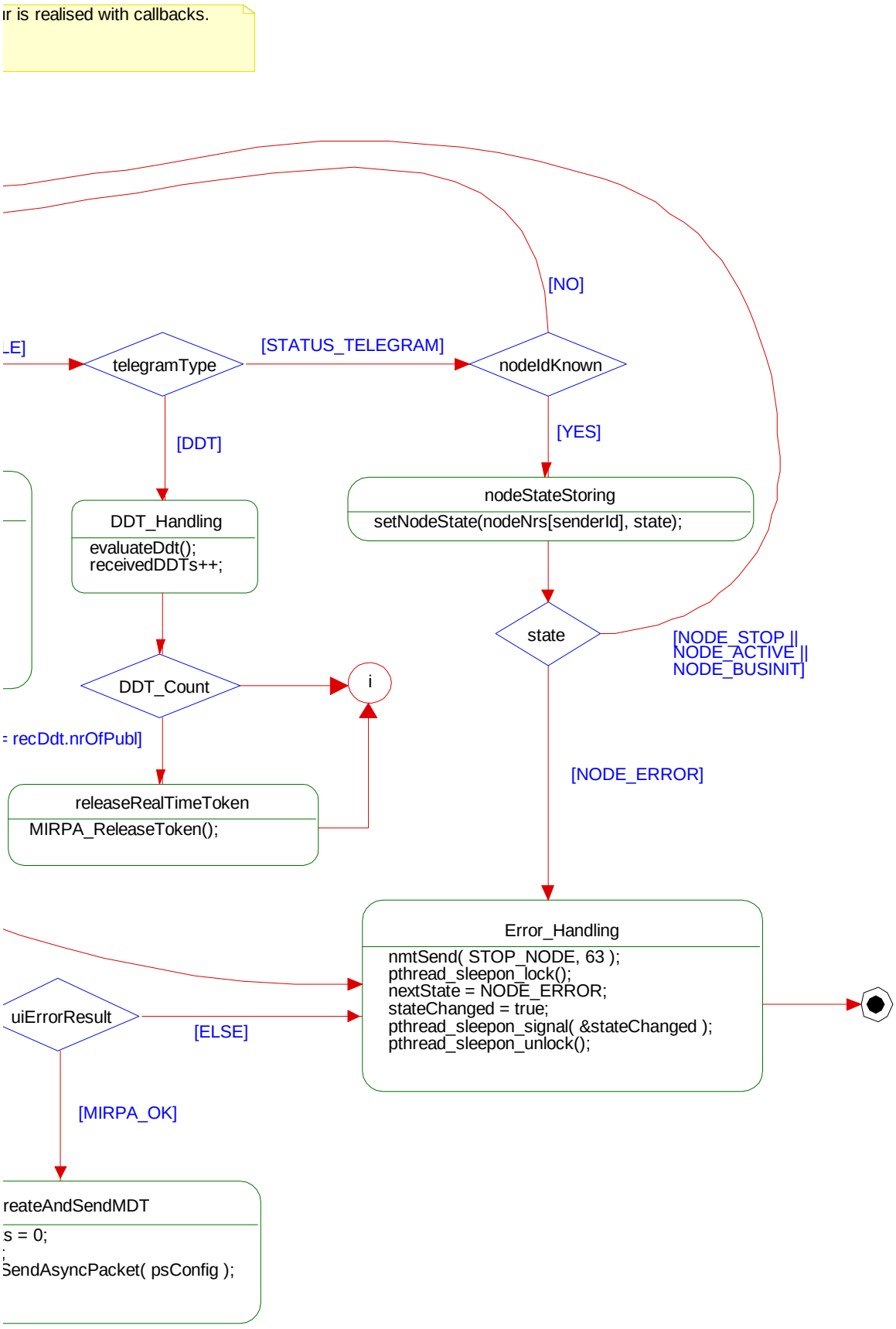



Abbildung 4.42: Aktivitätsdiagramm des Masters im Zustand ACTIVE


ır is realised with callbacks.
[NO]
LE]
telegramType
[STATUS_TELEGRAM]
nodeIdKnown
[DDT]
[YES]
nodeStateStoring
setNodeState(nodeNrs[senderId], state);
DDT_Handling
evaluateDdt();
receivedDDTs++;
state
[NODE_STOP ||
NODE_ACTIVE ||
NODE_BUSINIT]
DDT_Count
i
= recDdt.nrOfPubl]
releaseRealTimeToken
MIRPA_ReleaseToken();
[NODE_ERROR]
Error_Handling
nmtSend( STOP_NODE, 63 );
pthread_sleepon_lock();
nextState = NODE_ERROR;
stateChanged = true;
pthread_sleepon_signal( &stateChanged );
pthread_sleepon_unlock();
uiErrorResult
[ELSE]
[MIRPA_OK]
reateAndSendMDT
s = 0;
SendAsyncPacket( psConfig );

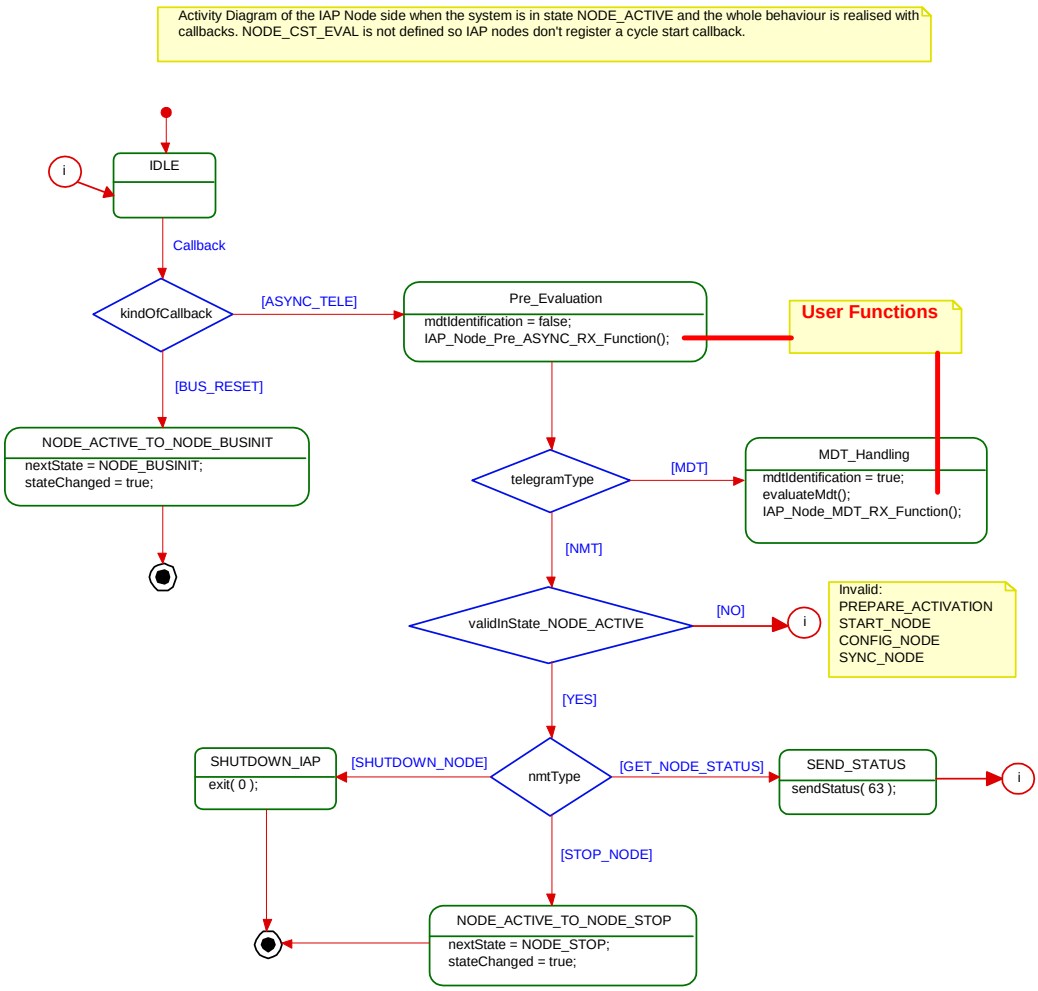



Abbildung 4.43: Aktivitätsdiagramm eines Knotens im Zustand ACTIVE

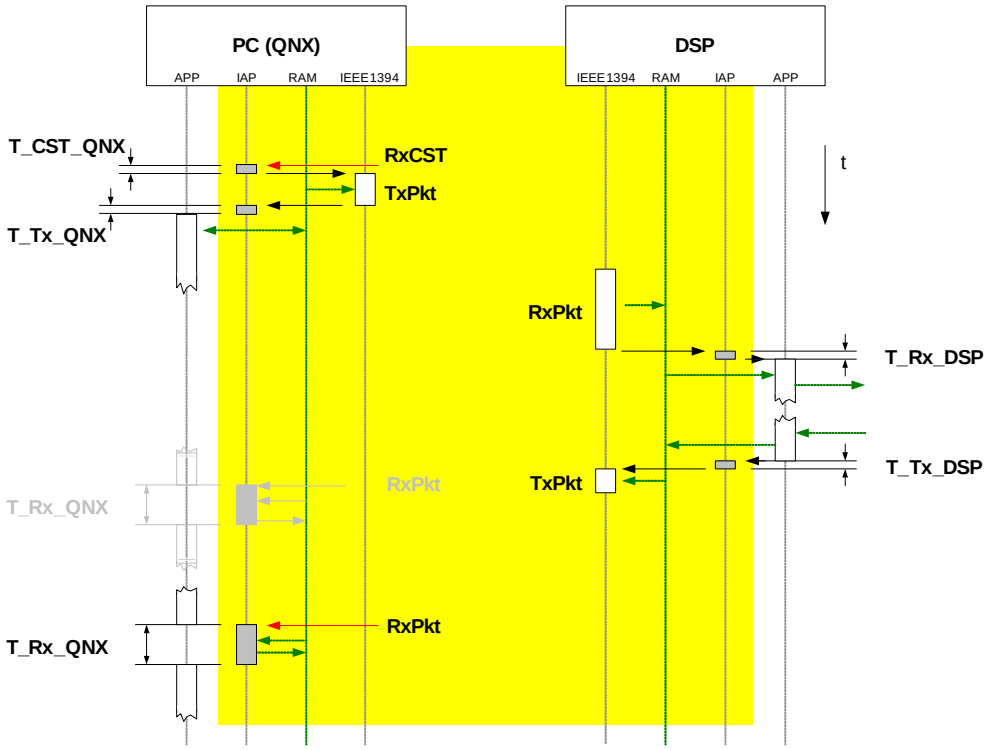



Abbildung 4.44: Zyklischer Ablauf und Leistungsdaten des IAP

vorliegen, wird innerhalb einer vom Treiber angestoßenen IAP-Auswertefunktion das MDT registriert und der Zeiger auf den für den Knoten relevanten Wertebereich an die Applikation (z.B. Antrieb) weitergegeben. Dies geschieht während der Zeit `T_Rx_DSP`. Anschließend erfolgt die Datenverarbeitung durch die Applikation, die innerhalb der Zeit `T_Tx_DSP` mit dem Zusammenstellen des DDT abgeschlossen wird. Der IEEE1394-Treiber versendet das Telegramm.

Der DMA-Transfer auf dem PC erledigt das Einlesen der einkommenden DDT. Über den IEEE1394-Treiber erfolgt eine Signalisierung an das IAP, sobald eine erwartete Anzahl Telegramme bzw. eine entsprechende Datenmenge eingetroffen ist. Bei einem auch möglichen Einzelempfang der Telegramme würde sich die zur Auswertung der DDT benötigte Zeit `T_Rx_QNX` entsprechend vervielfachen. Während der Auswertung der DDT geht das IAP jeden Telegrammkopf durch und ordnet die Daten den jeweiligen Shared-Memory-Bereichen (gemäß der Einstellungen in der XML-Konfigurationsdatei) zu. Dorthin werden die Nutzdaten auch entsprechend kopiert und anschließend die Applikationsprozesse ausgeführt, die ihrerseits die Verarbeitung der Daten vornehmen. Konnte der Empfang aller DDT nicht in der vereinbarten Zeit festgestellt werden, so erfolgt eine Fehlermeldung, die steuerungsseitig über eine geeignete Funktionalität behandelt werden muss.

Zur Messung der in Abbildung 4.44 angegebenen Zeiten wurden Datentelegramme mit einer Größe von jeweils 20 Byte übertragen. Da während des zyklischen Betriebs innerhalb IAP nur Datenzeiger übermittelt werden, sind die Ausführungszeiten dafür unabhängig von der tatsächlich übertragenen Datenmenge. Tabelle 4.1 zeigt die

| Zeitparameter | Zeit in $\mu s$ |
|---|---|
| T_CST_QNX | 4 |
| T_Tx_QNX | (4) |
| T_Rx_DSP | 4 |
| T_Tx_DSP | 4 |
| T_Rx_QNX | 10 |

Tabelle 4.1: Gemessene Ausführungszeiten von IAP-Funktionen

ermittelten Zeitwerte für unterschiedliche IAP-Aktivitäten.

Die realisierten Zeiten zeigen, dass die zyklische Belastung der Verarbeitung über das IAP in einem vertretbaren Maß bleibt. Verglichen mit den Zeitangaben für Synchronisierung und Scheduling des MiRPA-X-Prozesses (s. Absatz 4.2.3) liegen sie in den Dimensionen der Kontextwechsel auf Applikationsebene. Ausgehend von einer angestrebten Zykluszeit von nur $125\mu s$ fallen die Aktivitäten des Protokolls damit allerdings unter die Hauptverbraucher des Systems.

#### Optimierung des Echtzeitdatentransfers

In Abschnitt 4.2.2 ist der Vorgang zur Konfigurierung von EXTERN-Shared-Memory innerhalb der Anwendung von MiRPA-X beschrieben (s. 114). Über das IAP wird zur Optimierung des Datendurchsatzes auf dem zentralen Steuerungsrechner die Verknüpfung des IEEE1394-Blocktransfers und der erwähnten EXTERN-Shared-Memory-Funktionalität realisiert.

Abbildung 4.45 zeigt die Prozessstruktur, die bei der Anwendung des EXTERN-Shared-Memory verwendet wird, um die zunächst unsortiert einkommenden Datentelegramme in kürzester Zeit den jeweiligen Applikationsprozessen zugreifbar zu machen. Die Erläuterungen in diesem Abschnitt ergänzen dabei die Ausführungen zur Abbildung 4.22. Zum besseren Verständnis der Threadzusammenhänge, die in Abbildung 4.45 nur angedeutet sind, sei auf Abbildung 4.38 verwiesen.

Nachdem das IAP die XML-Konfigurationsdatei eingelesen, den Ziel-Speicherblock für die einkommenden Telegramme vom IEEE1394-Treiber erfragt und über MiRPA-X für diesen Speicherblock eine EXTERN-Shared-Memory-Gruppe eingerichtet hat (0), wird der Speicherblock über die MiRPA-X-API partitioniert und die Applikationsprozesse reservieren ihren Zugriff darauf. Während des zyklischen Betriebs im aktiven Zustand des IAP sorgen folgende Operationen für eine korrekte Zuordnung der einkommenden Daten zu entsprechenden Applikationsprozessen:

**1** Nachdem über die IEEE1394-Hardware per DMA ein vollständiger Telegrammblock im dafür vorgesehenen Datenspeicher empfangen wurde, signalisiert der IEEE1394-Treiber dem IAP die Ankunft. Dieses Ereignis startet den Auswertevorgang.

**2** Das IAP geht nun jeweils die DDT-Header in allen empfangenen Telegrammen

durch, prüft ihre Absender und merkt sich den Anfang (Offset) der jeweiligen Datenbereiche innerhalb des Empfangsspeichers.

**3** Die ermittelten Offsets werden gemäß eines über die Sender-Identifikationsnummer (Knoten-ID) eindeutig zugeordneten Index in die Gruppen-Verwaltungsstruktur eingetragen. Anschließend wird die Aktivität des IAP für diesen Zyklusteil beendet, so dass nun die Token-Threads der Applikation zur Ausführung kommen. 

**4** Durch eine für den Anwender unsichtbare Funktionalität innerhalb der MiRPA-X-API wird beim Eintreffen des Tokens in den Applikationsprozessen zunächst der DDT-Datenzeiger der Applikation gemäß des jeweiligen Offsets modifiziert. Erst dann erfolgt die Ausführung der Applikationsfunktionalität, in der die nun korrekten Daten ausgelesen werden können.

Diese Daten und die zugehörigen Offsets sind für den aktuellen Zyklus (bis zum Eintreffen neuer DDT) gültig. Die empfangenen Daten stellen also über die Aktivitäten des IAP immer ein aktuelles Abbild der Prozessdaten dar. Sollten Werte über einen Zyklus hinaus unverändert bleiben, so sind diese innerhalb des Zyklus an einer anderen Stelle außerhalb des EXTERN-Shared-Memory-Bereichs zu sichern.

## 4.4 Kommunikationssystem IEEE1394

Als Grundlage für alle externen Kommunikationsaktivitäten von der zentralen Steuerung zu den Aktoren und von den Sensoren der verschiedenen Roboter-Teilanwendungen hin zur Steuerung wurde im Rahmen der hier vorliegenden Arbeit ein für die Anwendung innerhalb des SFB 562 zugeschnittenes Kommunikationssystem realisiert, welches auf dem IEEE1394-Standard basiert. Dieser leistungsstarke Kommunikationsstandard bestimmt damit die Art und Weise, in der sämtliche physikalische Übertragung von Datentelegrammen innerhalb der praktischen Anwendung vor sich geht.

Kommerziell verfügbare Realisierungen der dazu benötigten Kommunikationsmodule (z.B. PC-Einsteckkarten) entsprachen zum Zeitpunkt der Konzipierung entweder nicht den Leistungsanforderungen innerhalb des SFB 562 oder ihre Treibersoftware war zusätzlich auch noch für unpassende Betriebssysteme implementiert. Trotz dieses Umstands wurden zunächst die Applikationen der Firmen Mindready (für die PC-basierte Hard- und Software) und Orsys (für die mikrocontrollerbasierte Hard- und Software) verwendet, um eine erste Realisierung des Kommunikationssystems zu entwickeln [132]. Die dabei gewonnenen grundlegenden Erfahrungen mit der Verwendung dieses Kommunikationssystems führten im weiteren Projektfortschritt zur Entwicklung eigener Kommunikationshardware auf Basis vorhandener integrierter Schaltkreise für Physical- und Link-Layer-Controller.

Aufsetzend auf der für die Kommunikationshardware ebenfalls innerhalb der vorliegenden Arbeit entworfenen und softwaretechnisch umgesetzten Treiberebene kommt

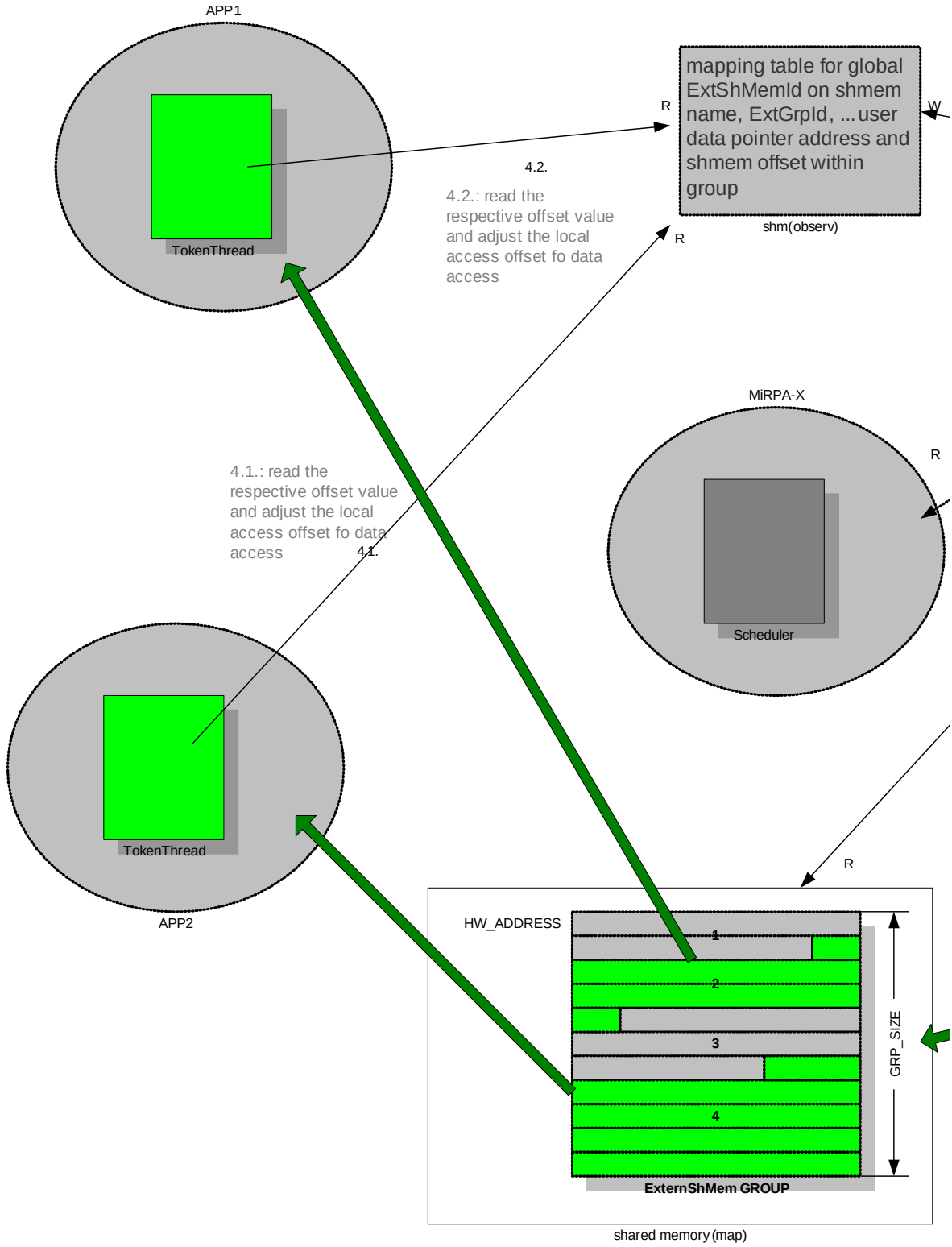
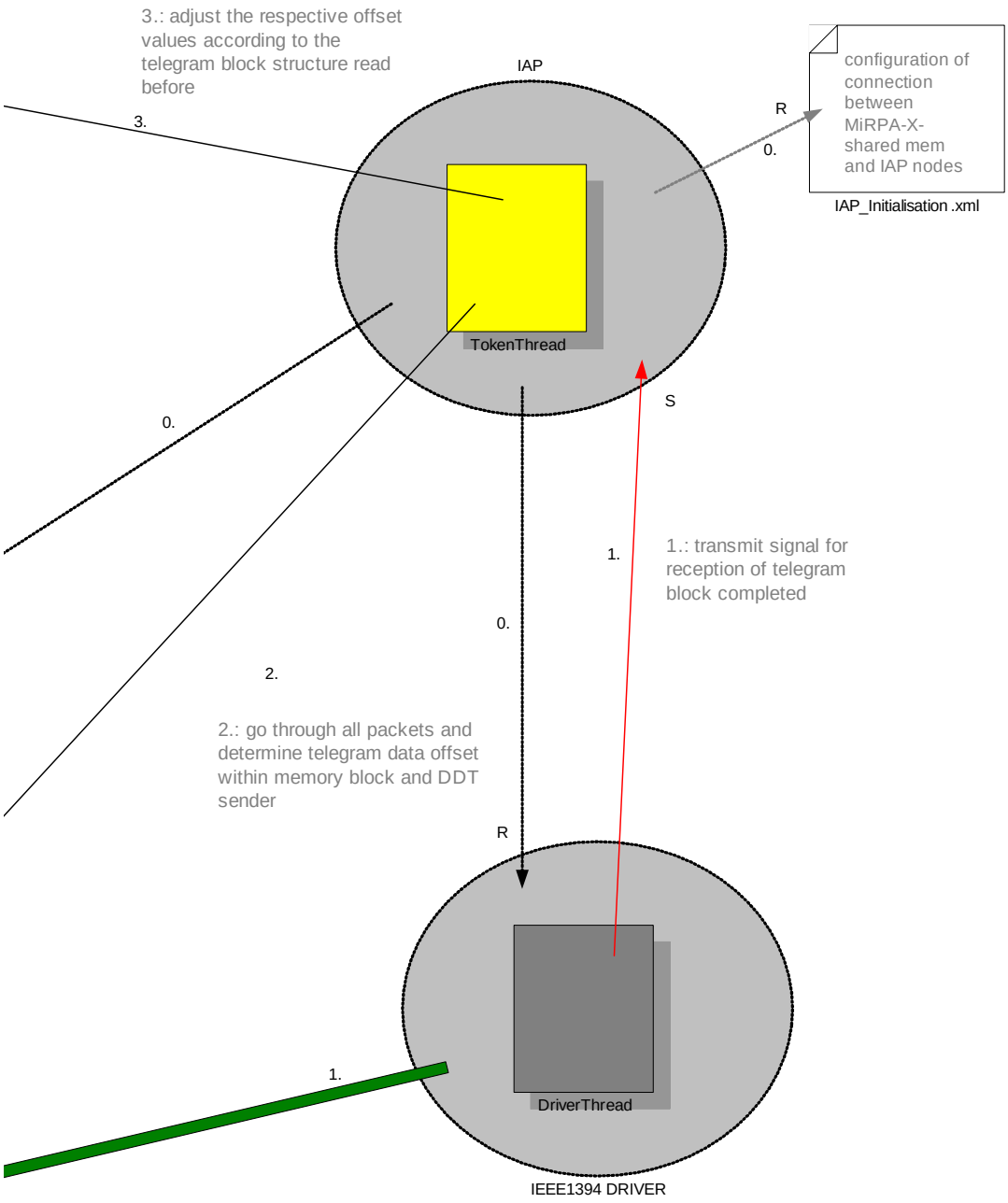



Abbildung 4.45: Nutzung der EXTERN-ShMem-Funktionalität


3.: adjust the respective offset values according to the telegram block structure read before
3.
IAP
R
0.
configuration of connection between MiRPA-X-shared mem and IAP nodes
IAP_Initialisation .xml
TokenThread
0.
S
1.
1.: transmit signal for reception of telegram block completed
0.
2.
2.: go through all packets and determine telegram data offset within memory block and DDT sender
R
1.
DriverThread
IEEE1394 DRIVER

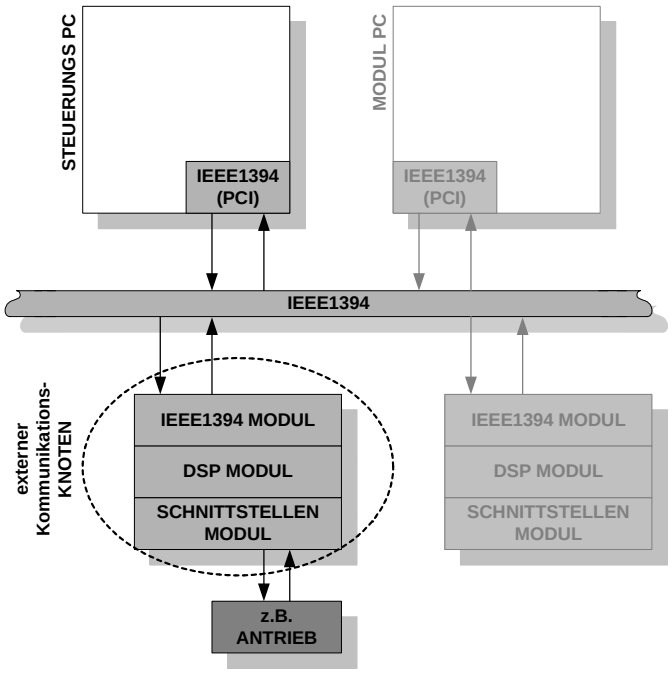



Abbildung 4.46: Schema der Kommunikationshardware

das IAP (s. Abschnitt 4.3) zur Komponentenkonfiguration und zum Netzwerkmanagement zum Einsatz. Bei der Realisierung dieser Kommunikationsschnittstelle war zu beachten, dass der Soft- und Hardwarezugriff zum einen von der Steuerungsseite und zum anderen von der Sensor/Aktor-Seite her realisiert werden musste. Die notwendigen Funktionalitäten sowie eine für beide verwendeten Plattformen (PC und DSP) einheitliche Zugriffs-Syntax war zu definieren und umzusetzen. Die Funktionsweise und Realisierung der IEEE1394-Kommunikationsschnittstelle wird im Folgenden beschrieben.

### 4.4.1 Aufbau des Kommunikationssystems

Abbildung 4.46 zeigt eine schematische Übersicht über den Hardwareaufbau des in dieser Arbeit entwickelten Kommunikationssystems. Das Kommunikationsmedium ist der IEEE1394-Bus, wie er in Abschnitt 3.3.1 vorgestellt wurde. Die Prozesse auf einem zentralen Steuerungs-PC greifen über eine IEEE1394-PCI-Karte auf das Kommunikationsmedium zu und nehmen damit am externen Kommunikationsgeschehen teil. Die externen Roboterkomponenten (z.B. Antriebe) tauschen über jeweils ein Schnittstellenmodul Daten mit einem verarbeitenden DSP-Modul (externe Recheneinheit) aus. Dieses ist in der Lage, Kommunikationsdaten über ein IEEE1394-Modul auf den Bus zu schreiben oder von dort zu lesen. Diese drei Module werden zu jeweils einem „externen Kommunikationsknoten“ zusammengefasst und als Platinenstapel für das jeweilige anzuschließende Gerät zur Verfügung gestellt.

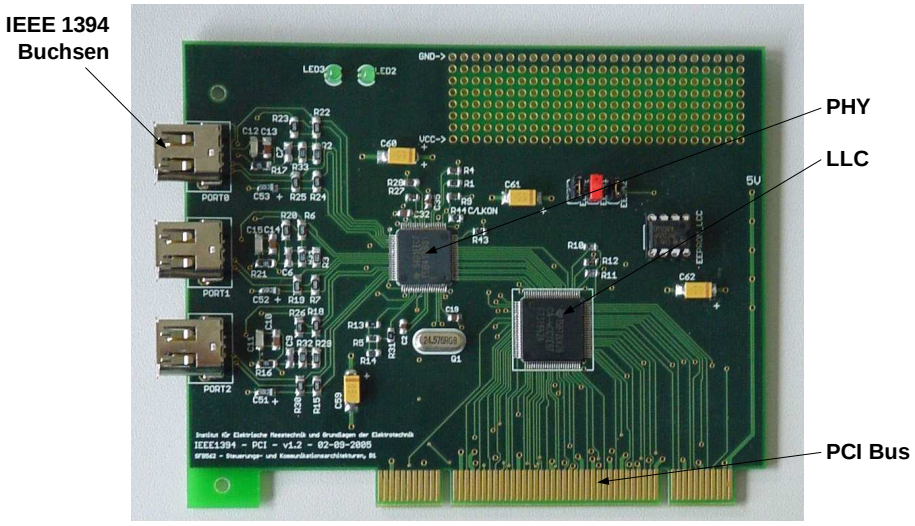



Abbildung 4.47: IEEE1394-Hardwarerealisierung für PCI

### 4.4.2 Realisierung

Ausgehend von der Darstellung des Aufbaus der entwickelten Hardwaremodule der IEEE1394-Schnittstelle erfolgt in diesem Abschnitt die Beschreibung der wesentlichen Funktionen zu ihrem praktischen Einsatz. Dabei wird immer zwischen der Anwendung auf dem PC und der entsprechenden DSP-Realisierung unterschieden; auf sie wird separat eingegangen.

#### Hardware

Abbildung 4.47 zeigt die entwickelte PCI-Karte, die innerhalb des externen Kommunikationssystems PC-seitig zum Einsatz kommt. Hier werden, wie dargestellt, Standard-Bauelemente und jeweils eine aktuelle Variante des Link-Layer-Controllers (LLC, TSB12LV26) und des Physical-Layer-Controllers (PHY, TSB41AB3) für den IEEE-1394a-Standard verwendet. Diese Bauelemente ermöglichen einen Datendurchsatz für das Übertragen jeder möglicher Paketform in das LLC-FIFO und aus ihm heraus mit der im Standard spezifizierten Übertragungsrate von $400 Mbit/s$.

Die Funktionalitäten des verwendeten LLC können softwareseitig über den OHCI[3]-Standard angesprochen werden, was sicherstellt, dass der Chip-Zugriff auch bei LLC neuer Bauart kompatibel bleibt. Außerdem beinhaltet der LLC bereits PCI-Funktionalität, was einen späteren Treiberentwurf stark vereinfacht, da Betriebssystemfunktionen verwendet werden können, die die implementierten Funktionen zur PCI-Organisation und PCI-Synchronisation direkt ansprechen. Der LLC bietet die Möglichkeit, Nutzdaten per DMA in das RAM des PC und von dort heraus zu übertragen, dadurch minimiert sich der für den Prozessor anfallende Zeitaufwand. Spezielle Einstellmöglichkeiten erlauben das Verwenden des LLC im Blockbetrieb, der den beson-

[3]Open Host Controller Interface

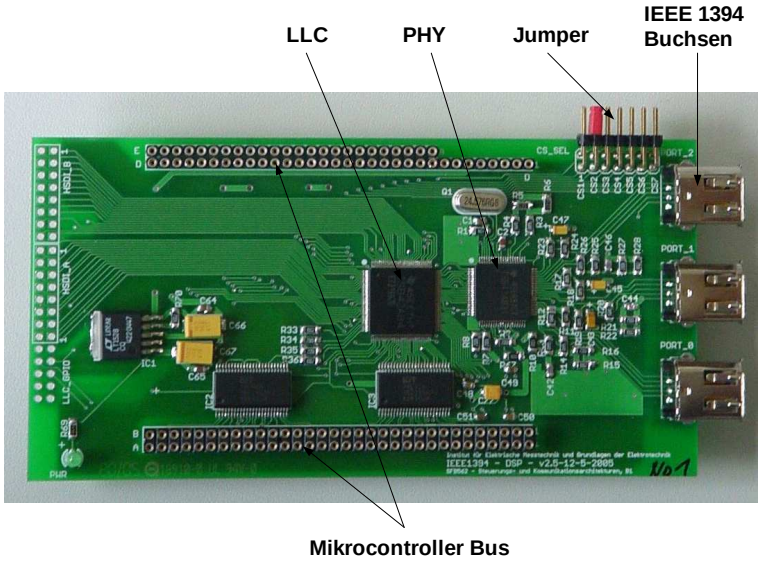



Abbildung 4.48: IEEE1394-Hardwarerealisierung für DSP-Module

deren Vorteil bietet, eine Interrupt-Benachrichtigung an den Prozessor erst dann zu generieren, wenn eine bestimmte Anzahl von Telegrammen oder eine entsprechende Datenmenge empfangen wurde. Von dieser Möglichkeit wird bei der Realisierung des IAP Gebrauch gemacht (s. Abschnitt 4.3.3).

Der verwendete PHY ist nach dem IEEE1394a-2000-Standard realisiert und bietet drei voneinander unabhängige Ports zum Anschluss weiterer IEEE1394-Geräte. Zum Betrieb wird ein externer $24.576 MHz$-Referenzquarz benötigt, aus dem auch der intern verwendete Takt von $393.216 MHz$ per PLL und der vom LLC benötigte $49.125 MHz$-Takt abgeleitet wird.

Abbildung 4.48 zeigt die entwickelte IEEE1394-Karte zum Einsatz innerhalb eines externen Kommunikationsknotens. Sie besitzt ebenfalls drei Ports und enthält neben Standard-Bauelementen und einem auf die Anschlüsse des DSP-Moduls angepassten Mikrocontroller-Bus auch Bustreiber, um die Leistung der digitalen Signale zu verstärken und damit Kartenerweiterungen mit langen Busleitungen zu ermöglichen. Über Jumper an der Oberkante der Karte lassen sich die von der Karte verwendeten Interruptleitungen konfigurieren, um auf diese Weise Konflikte mit anderen Karten zu vermeiden.

Während der PHY identisch zu dem auf der PCI-Karte ist, wird hier ein TSB42AB4 („ceLynx“) als LLC verwendet, den man zur Anwendung für verschiedene Mikrocontroller-Busse oder einen HSDI[4] konfigurieren kann. Er bietet einen $8 kByte$ großen Sende- und Empfangsspeicher (FIFO), der individuell in acht unabhängig konfigurierte Bereiche unterteilt werden kann, um die Leistungsfähigkeit an die jeweilige Anwendung anpassen zu können.

[4]High Speed Data Interface

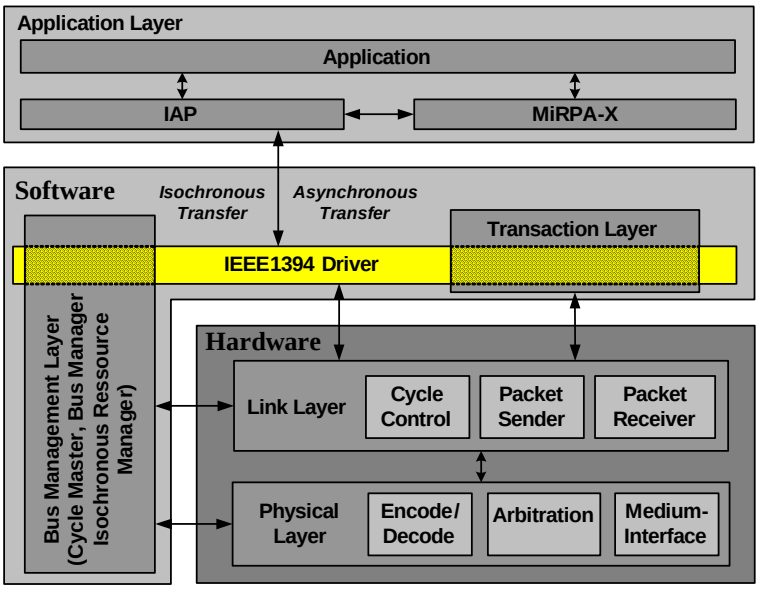



Abbildung 4.49: IEEE1394-Treiber innerhalb der Softwareschichten

### Treibersoftware

Die dargestellte Hardware wird jeweils PC- und knotenseitig über spezifisch dafür entwickelte Treibersoftware gesteuert. Gemeinsam mit dem verwendeten Kommunikationsprotokoll IAP stellen diese Soft- und Hardwareschichten die Kommunikationsebene für die Ansteuerung externer Sensoren und Aktoren durch eine Applikation bereit. Das Schichtenmodell hierfür ist für die PC-Anwendung in Abbildung 4.49 dargestellt. Bei der Anwendung auf dem DSP fehlt lediglich die MiRPA-X-Schicht, da aufgrund des dort nicht verfügbaren Betriebssystems keine Anwendung von Prozessen oder Threads möglich und damit auch nicht zu verwalten ist.

In Abbildung 4.50 ist die Software-Architektur des IEEE1394-Treibers für die Anwendung auf einem PC unter dem Betriebssystem QNX dargestellt. Sie ist in drei Bestandteile aufgeteilt (Anwenderprozess mit API-Nutzung, gemeinsam genutzter Daten- und Statusspeicher und Treiberprozess). Die Aufteilung in zwei unabhängige Prozesse ist sinnvoll, da so unter QNX mehr Flexibilität und Modularität entsteht, die gerade bei komplexen Anwendungen erforderlich sind.

Im oberen Teil der Abbildung ist der eigentliche Hardwaretreiber (driver process) dargestellt. Er stellt als Serverapplikation die Verbindung zwischen dem Nutzer und den auf dem Bus kommunizierten Daten her. Der Treiberprozess besteht aus zwei Arbeitsthreads (InterruptThread und ResourceThread), die jeweils für den einkommenden bzw. ausgehenden Datenstrom verantwortlich sind. Sie kommunizieren untereinander über den Prozessspeicher und nach außen hin über Interrupt-Signale, Nachrichten und Semaphoren, wie im Folgenden erläutert wird:

**interrupt thread** blockiert auf das Eintreffen einer Benachrichtigung über den Interrupt-Pin des LLC, überprüft das Ereignis und ruft eine entsprechende Callback-

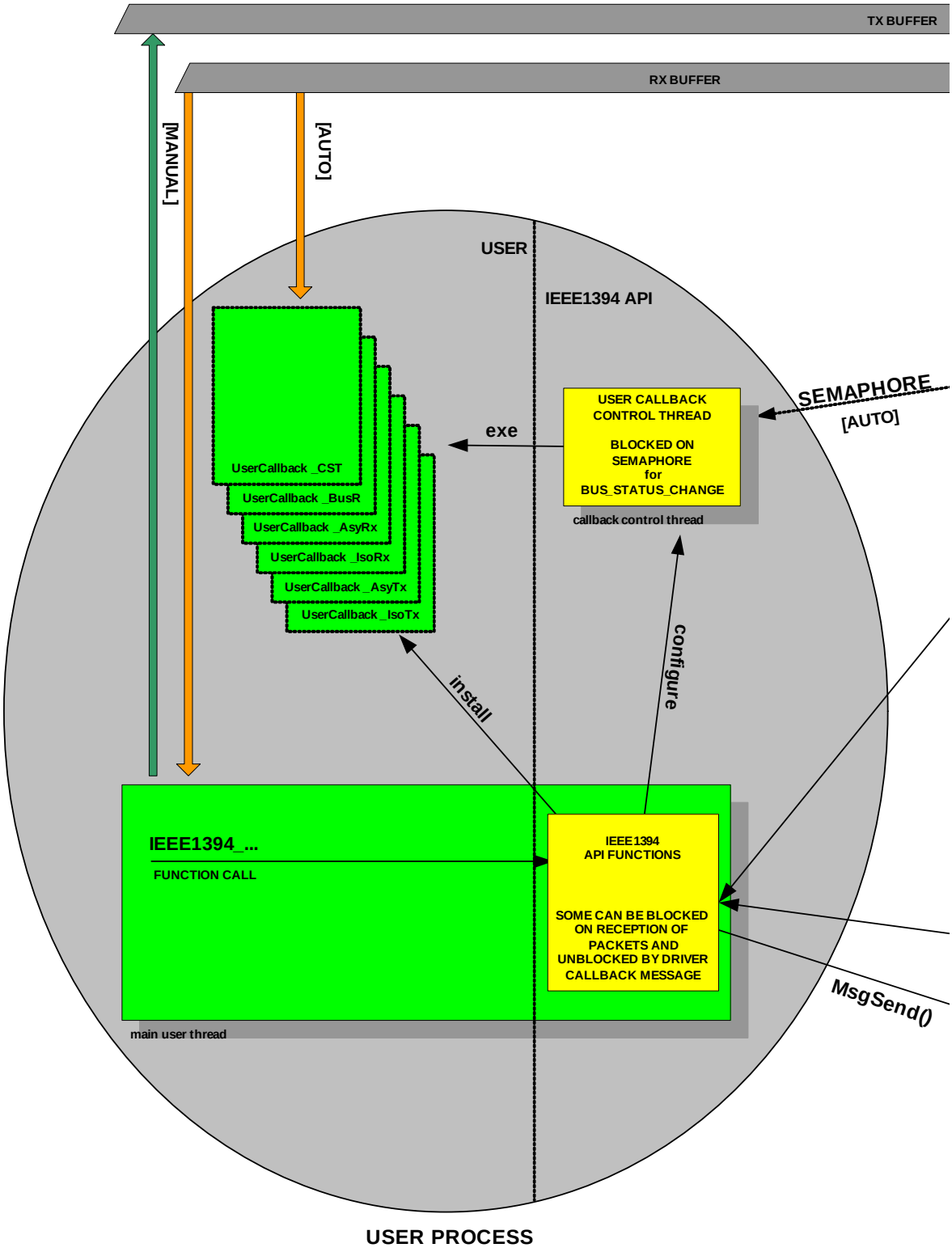
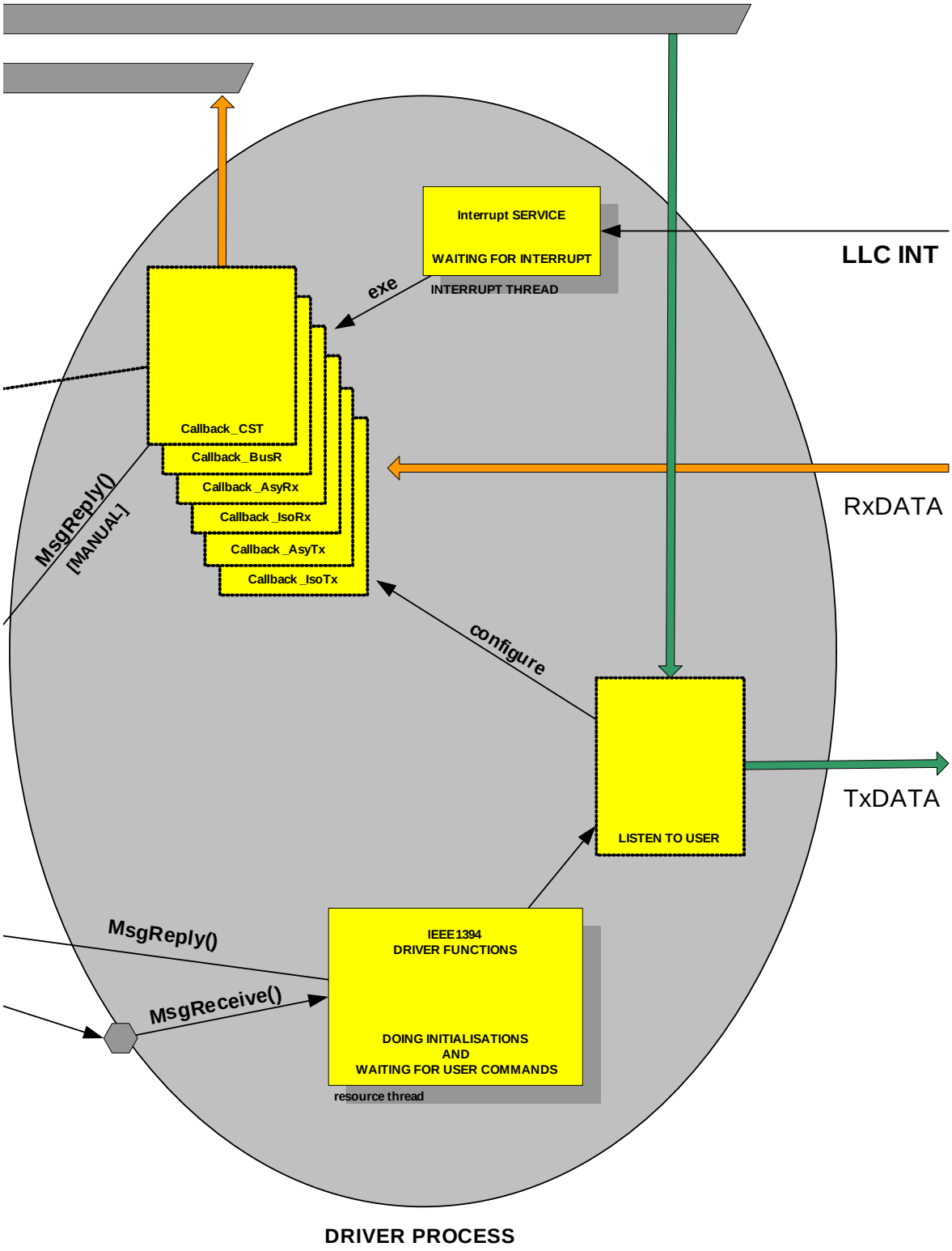



Abbildung 4.50: Softwarearchitektur der IEEE1394-Anwendung für QNX


Interrupt SERVICE
WAITING FOR INTERRUPT
INTERRUPT THREAD
exe
LLC INT
Callback_CST
Callback_BusR
Callback_AsyRx
Callback_IsoRx
Callback_AsyTx
Callback_IsoTx
MsgReply()
[MANUAL]
RxDATA
configure
TxDATA
LISTEN TO USER
MsgReply()
MsgReceive()
IEEE1394
DRIVER FUNCTIONS
DOING INITIALISATIONS
AND
WAITING FOR USER COMMANDS
resource thread
DRIVER PROCESS

funktion auf, die die einkommenden Daten für den Anwender vorbereitet. Die Benachrichtigung des Anwenderprozesses kann auf zwei verschiedene Weisen erfolgen. Im Fall eines *manuellen* Empfangsbetriebs erfolgt sie über eine Antwortnachricht (MsgReply) an den anfragenden Anwenderprozess (MsgSend); bei *automatischem* Empfangsbetrieb dagegen über die Synchronisierung mittels einer Semaphore, die im Anwenderprozess weitere Aktivitäten triggert, die weiter unten beschrieben werden.

**driver resource thread** blockiert auf das Eintreffen einer Nachricht (MsgReceive) vom Anwenderprozess und wertet diese beim Eintreffen aus. Bei Bedarf werden die Initialisierung der Hardware sowie Einstellungen für den Datenempfang vorgenommen. Eine Sendeanforderung vom Anwenderprozess erfolgt direkt über das Setzen eines LLC-Startbits, welches bewirkt, dass sich der LLC per DMA die Ausgangsdaten greift und auf den Bus schreibt. Ein erfolgreiches Schreiben auf den Bus wird über eine Interrupt-Benachrichtigung vom LLC beantwortet und kann von dort aus an den Anwenderprozess bestätigt werden.

Auch das Eintreffen von Zyklus-Starttelegrammen (CST) und Fehlertelegrammen wird auf diese Weise über den LLC benachrichtigt und an den Anwenderprozess weitergeleitet, der dann basierend auf (auch zyklischen) Busvorgängen eigene Funktionalitäten realisieren kann.

Im unteren Teil der Abbildung ist der Anwenderprozess (user process) dargestellt. Er stellt als Clientapplikation die Verbindung zwischen den Anwendungsfunktionen (Nutzdaten und Ablauf) und dem IEEE1394-Treiber her. Der Anwenderprozess besteht ebenfalls aus (mindestens) zwei Arbeitsthreads (MainUserThread und CallbackControlThread), die einerseits die Programmierschnittstelle (API) entsprechend der Anwendung bedienen und andererseits die vom Bus her kommenden Ereignisse mit der Ausführung der vom User gewünschten Funktionalitäten verbinden:

**main user thread** greift über die Ausführung von `IEEE1394_...`-Funktionsaufrufen auf die API-Funktionen zu, die wiederum die Initialisierung der Anwenderfunktionen, die Konfigurierung der applikationsspezifischen Ereignisauswertung und die Nachrichtenkommunikation mit dem Treiberprozess erledigen. Die Kommunikation mit dem Treiber erfolgt innerhalb der API über blockierende QNX-Nachrichten, so dass die Abgeschlossenheit von kritischen Vorgängen innerhalb des Treibers sichergestellt ist, bevor Funktionalitäten innerhalb des Anwendungsprozesses auf entsprechende Daten zugreifen können. Dabei geschieht die Beantwortung der Nachrichtenanfragen je nach angestoßenem Vorgang aus dem ResourceThread oder dem InterruptThread des Treiberprozesses heraus.

**callback control thread** wird beim Aufruf der Funktion `IEEE1394_Init()` automatisch erzeugt und wartet nach der Konfigurierung durch die API auf das Setzen festgelegter Semaphoren vom InterruptThread des Treiberprozesses. Auf diese Synchronisationsereignisse hin werden dann die entsprechenden Anwenderfunktionalitäten, die innerhalb von UserCallbacks realisiert und zuvor über API-Funktionen installiert wurden, zur Ausführung gebracht.

Die beiden sich quer über die Prozesse erstreckenden Balken innerhalb der Abbildung symbolisieren Shared-Memory-Bereiche, die zur Kommunikation von Status- und Nutzdaten zwischen den Prozessen verwendet werden. Auf diese Weise kann ein zeitraubendes Kopieren von Nutzdaten zwischen den Softwareebenen verhindert und damit die Verarbeitungsgeschwindigkeit optimiert werden. Auf den Bus zu schreibende bzw. hiervon zu lesende Daten werden in Datenpuffern abgelegt, die über eine Referenzadresse beiden Applikationen bekannt sind. Der Empfangsspeicher (rx buffer) ist als FIFO-Speicher organisiert, so dass schnell einkommende Daten ältere Daten nicht überschreiben können. Den automatisch über Semaphor-Signalisierung und den manuell über Nachrichtenversand aufgerufenen Benutzerfunktionen werden nur Referenzen auf die tatsächlichen Datenspeicher übergeben.

In Abbildung 4.51 ist die Software-Architektur des IEEE1394-Treibers für die Anwendung auf einem DSP-Modul ohne eigenes Betriebssystem dargestellt. Sie ist einfacher aufgebaut als im oben beschriebenen Fall, da es hier kein zu berücksichtigendes Betriebssystem gibt, das die Speicherzugriffe unterschiedlicher Prozesse regelt oder Kommunikations-Mechanismen zur Verfügung stellt, die die Interaktionen zwischen vielen gleichzeitig laufenden Prozessen verklemmungsfrei ermöglicht. Auf dem DSP läuft nur ein einziger Applikationsprozess, in den es alle Funktionalitäten modular einzuflechten galt. Dieser kann zu jeder Zeit von Interruptfunktionen (vom Timer und LLC) unterbrochen werden.

Die Treiberfunktionen sind innerhalb einer Funktionsbibliothek mit zugehöriger Header-Datei realisiert, die in die jeweilige Applikation eingebunden werden. Sobald der Benutzer die Init-Funktion aufruft, werden alle notwendigen Speicherbereiche (für Empfang, Senden und Status) reserviert und die für die Kommunikation mit der Hardware nötigen Interrupt-Service-Routinen und Konfigurationen eingestellt. Alle weiteren Funktionalitäten laufen nun entweder automatisch (bei Interrupt-Eingang) oder vom Benutzer initiiert (über C-Funktionen hervorgerufen) ab. Obwohl es keine unterschiedlichen Prozesse und Threads im eigentlichen Sinn gibt, können zwei quasi nebenläufige Vorgänge definiert werden:

**main user program** realisiert einen rein sequenziellen Ablauf und greift über `IEEE-1394_...`-Funktionen (API) auf die Kommunikationsschnittstelle zu. Hierdurch kann z.B. das Installieren von Nutzerfunktionen in den ereignisgesteuerten (interruptgesteuerten) Ablauf und das Konfigurieren der InterruptServiceRoutine zur Auswahl verwendeter Bus-Ereignisse erzielt werden. Neben den Busfunktionen müssen hier auch die Funktionen zur Kommunikation mit der Geräteschnittstelle aufgerufen werden, die beispielsweise Istwerte auslesen oder neue Sollwerte übertragen.

**interrupt service** wartet auf das Eintreffen eines Interrupts vom LLC und wertet diesen aus. Je nach Interruptquelle und vom Nutzer eingestellter Konfiguration kommen daraufhin die entsprechenden Callbacks zur Ausführung, in denen ggf. die Datenpuffer des LLC ausgelesen und dem Nutzer zur Verfügung gestellt werden. Von hier aus erfolgt dann der Aufruf der entsprechenden, vom Nutzer installierten Funktionalität innerhalb der UserCallbacks.

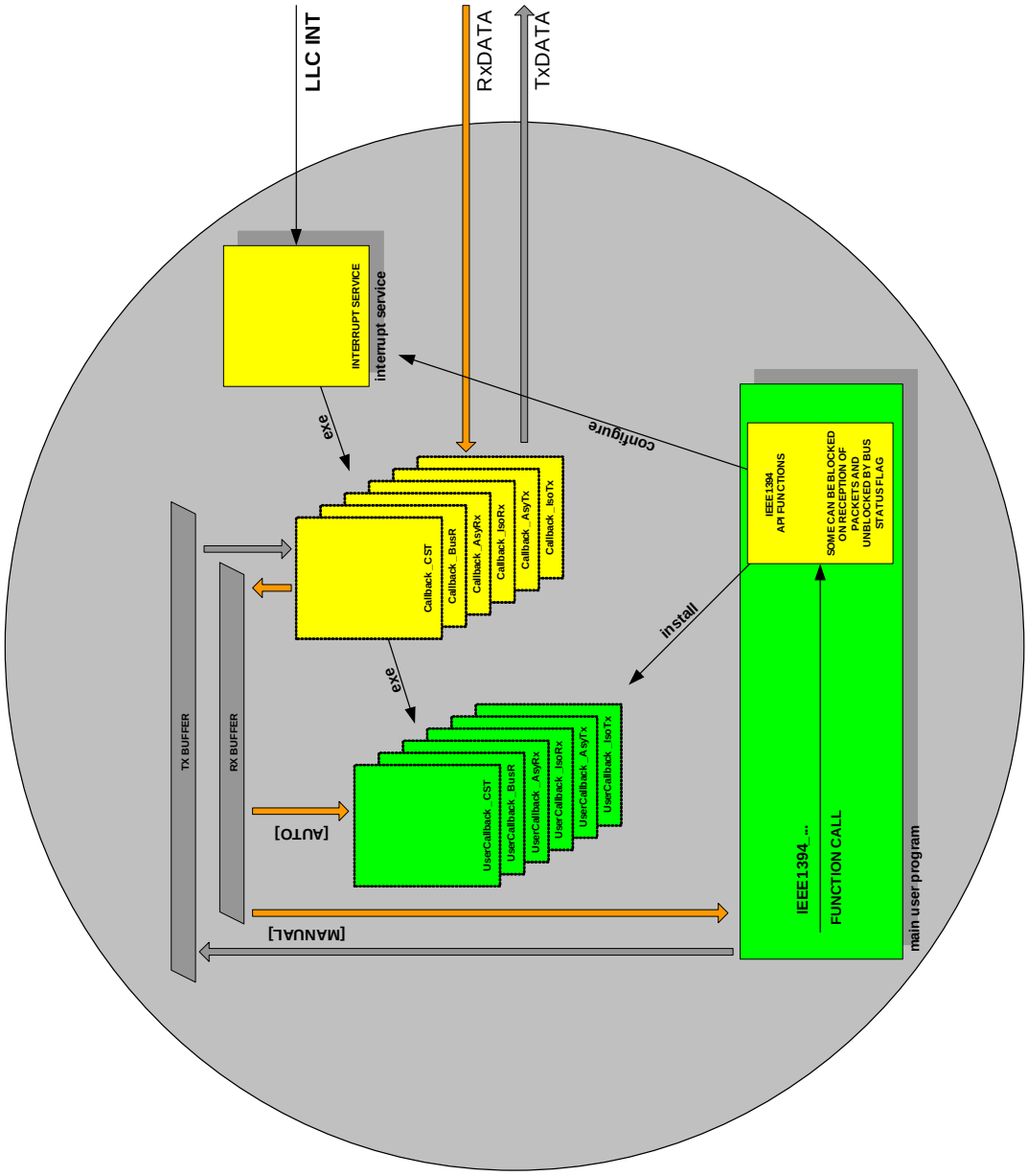



Abbildung 4.51: Softwarearchitektur der IEEE1394-Anwendung für DSP

| | |
|---|---|
| **IEEE1394_SetCallback()** | Festlegen einer Userfunktion, die ausgeführt werden soll, wenn ein Ereignis (BUS_RESET, CST, Rx, Tx,...) auftritt |
| **IEEE1394_ClearCallback()** | Löschen einer vorher festgelegten Userfunktion |
| **IEEE1394_SendAsyncPacket()** | Versenden eines asynchronen Telegramms über den IEEE 1394 bus |
| **IEEE1394_ReceiveAsyncPacket()** | Manuelles Empfangen eines asynchronen Telegramms vom IEEE 1394 bus |
| **IEEE1394_SendIsoPacket()** | Versenden eines isochronen Telegramms über den IEEE 1394 bus |
| **IEEE1394_ReceiveIsoPacket()** | Manuelles Empfangen eines isochronen Telegramms vom IEEE 1394 bus |

Abbildung 4.52: Ausschnitt aus der IEEE-1394-Programmierschnittstelle

### Programmierschnittstelle (API)

Bei der Realisierung der Programmierschnittstelle wurde darauf Wert gelegt, dass die jeweiligen Funktionsaufrufe, die unter ANSI C implementiert und angewendet werden, auf beiden Seiten (PC und DSP) identische Aktivitäten hervorrufen. Die Programmstruktur begünstigt die einfache Portierbarkeit auf weitere Plattformen. Abbildung 4.52 zeigt einen Ausschnitt aus der zur Verfügung gestellten Programmierschnittstelle (API). Alle für die Nutzung auf dem PC und DSP implementierten Funktionalitäten sind in dem zugehörigen Programmierhandbuch [135] ausführlich dokumentiert.

Alle den IEEE1394-Bus betreffenden Funktionen beginnen mit „`IEEE1394_`“, gefolgt von einem aufgabenbezogenen Funktionsnamen, der eine intuitive Nutzung der jeweiligen Funktionalität ermöglicht. Die Übergabeparameter sind in der Regel Daten- bzw. Funktionszeiger, die intern in Datenstrukturen eingetragen werden, um einen schnellen und automatischen Datenzugriff ohne überflüssige Speicherübertragungen zu ermöglichen. Als Rückgabeparameter verwendet die API Speicheradressen, die in Form von Übergabeparametern spezifiziert wurden und einheitliche Fehlercodes, die eine Ablaufsteuerung abhängig von Funktionserfolgen ermöglichen.

### Initialisierung und Start

Abbildung 4.53 zeigt das Sequenzdiagramm zur Initialisierung des IEEE1394-Treibers auf der PC-Seite. Die Initialisierung für den DSP ist im Prinzip ähnlich, aber einfacher aufgebaut und wird deshalb an dieser Stelle nicht weiter erläutert.

Nach dem Start des Treibers auf dem PC werden innerhalb des Hauptthreads (driver resource thread) einige Betriebssystem-Routinen aufgerufen, die eine neue PCI-Hardware (IEEE1394-Einsteckkarte) einrichten. Anschließend werden zum Betreiben der OHCI-Schnittstelle des LLC Speicherbereiche reserviert und die Konfigurierung des LLC vorgenommen. Ein MUTEX für die im Shared-Memory liegenden Busstatusvariablen schützt den Zugriff darauf. Zur Kommunikation mit dem Nutzerprozess wird ein Nachrichten-Kommunikationskanal eingerichtet und werden notwendige Vereinbarungen in einer temporären Datei gespeichert. In der Hauptschleife wird nun auf

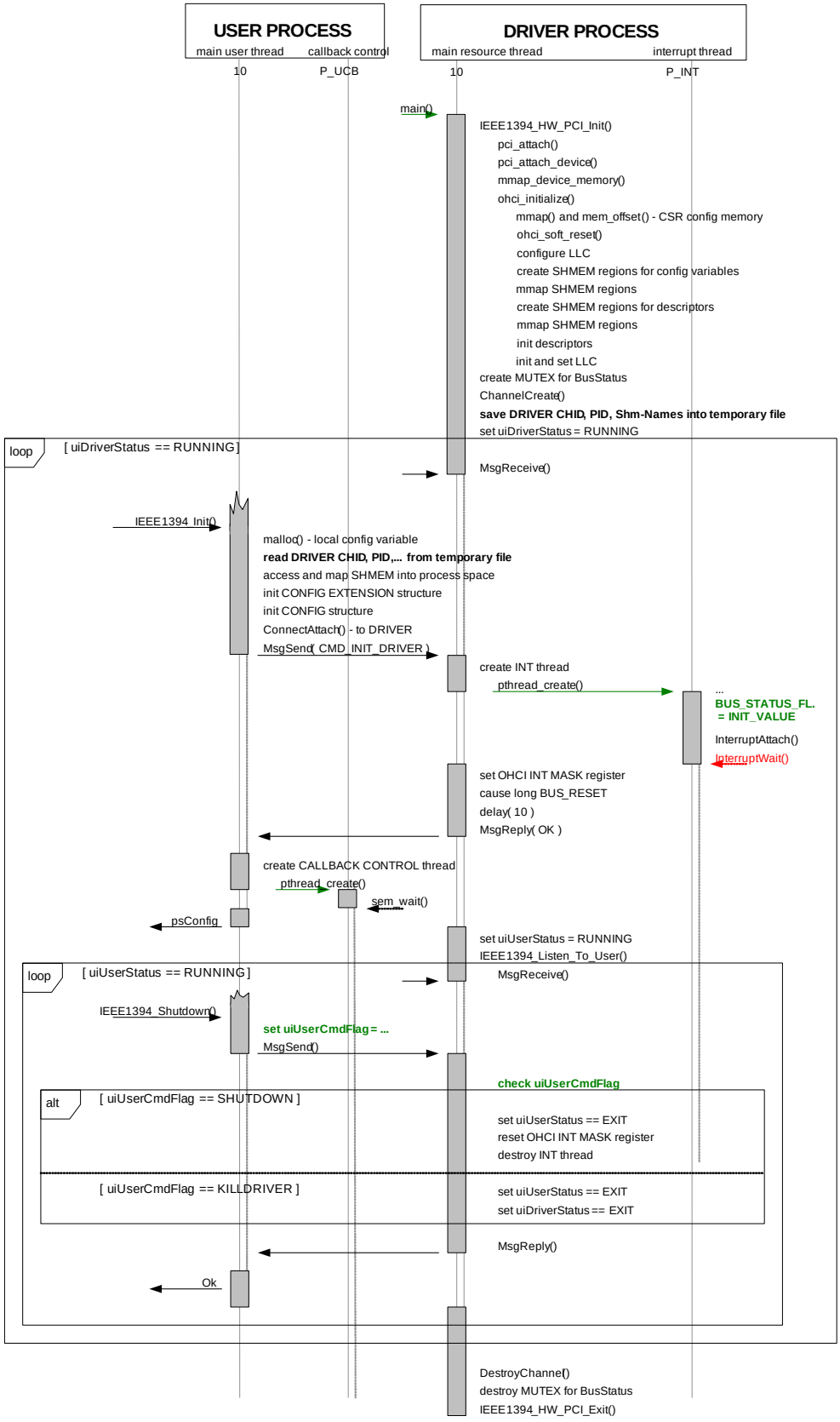



Abbildung 4.53: Vorgänge bei der Initialisierung der Hardware

das Eintreffen einer Nachricht vom Anwenderprozess gewartet.

Wenn nun innerhalb des Anwenderprozesses die Funktion `IEEE1394_Init()` aufgerufen wird, erfolgt das Reservieren lokaler Datenstrukturen, das Einlesen der zuvor vom Treiberprozess gespeicherten Vereinbarungen und das Einblenden benötigter Shared-Memory-Bereiche in den lokalen Prozessspeicher. Durch das Versenden eines Kommandos zum Fortsetzen der Treiber-Initialisierung wird mit der Ausführung des Treiberthreads fortgefahren, der nun weiß, dass ein Nutzerprozess Zugriff auf den Bus erlangen möchte. Innerhalb des neu erzeugten InterruptThreads werden nun alle Vorkehrungen getroffen, um das Empfangen und Auswerten des LLC-Interrupts durchzuführen.

Anschließend erfolgt das Beantworten der Nachricht, das innerhalb der API des Anwenderprozesses zum Erzeugen des Callback-Auswertungsthreads führt, der auf den Empfang einer Semaphore wartet. Innerhalb des Treibers sind nun alle Aktivitäten zur Initialisierung abgeschlossen. Dieser wartet nun auf das Eintreffen weiterer Nachrichten von dem Anwendungsprozess, um daraufhin den Buszugriff zu konfigurieren oder Daten abzuschicken. In Abbildung 4.53 sind auch noch die zwei Varianten zum Beenden des Treibers dargestellt, die aber hier nicht weiter erläutert werden.

### Block-Datenempfang

Um die typische zyklische Arbeitsweise des IEEE1394-Treibers im Zusammenhang mit dem Nutzerprogramm zu verstehen, werden in diesem Abschnitt die Vorgänge beim Empfang von Blocktelegrammen genauer beschrieben. Damit wird ein Einblick in den Sinn und die Arbeitsweise der unterschiedlichen Threads gegeben. Es wird nur die PC-Seite betrachtet, da die Vorgänge auf dem DSP zwar analog, allerdings wesentlich einfacher sind, da dort kein Block-Empfang realisiert werden musste.

In Abbildung 4.54 ist das Sequenzdiagramm für den automatischen Empfang von asynchronen Telegrammen im Blockmodus dargestellt. Beim automatischen Empfang von Telegrammen sind nur der Callback-Control-Thread des Anwendungsprozesses und der InterruptThread des Treiberprozesses beteiligt. Beide sind zunächst auf den Empfang von Ereignissen blockiert.

Empfängt der InterruptThread einen `ASYNC_RX_REQ`-Interrupt, so wird innerhalb der entsprechenden Auswertefunktion der eingestellte Empfangsmodus geprüft. Im Fall von `BLOCK_MODE` werden einige Voreinstellungen getätigt und innerhalb einer mutexgeschützten Schleife über alle empfangenen Telegramme des Blocks der jeweilige Transaktionscode `TCODE` ausgewertet. Für die Kommunikation innerhalb des IAP werden in der aktuellen Version ausschließlich asynchrone `BLOCK_REQUESTS` verwendet, so dass nur dieser Typ von Telegrammen innerhalb der Schleife weiter ausgewertet wird. Die Hauptaufgabe besteht nun darin, die Rahmeninformationen eines jeden Telegramms in einer Datenstruktur festzuhalten. Nach dieser Extraktion wird ein entsprechendes BusStatusFlag gesetzt, der Mutex freigegeben und eine Benachrichtigung über eine Semaphore abgesetzt. Die erneute Aktivierung des Datenempfangs geschieht durch das Zurücksetzen eines entsprechenden Bits der OHCI-Schnittstelle.

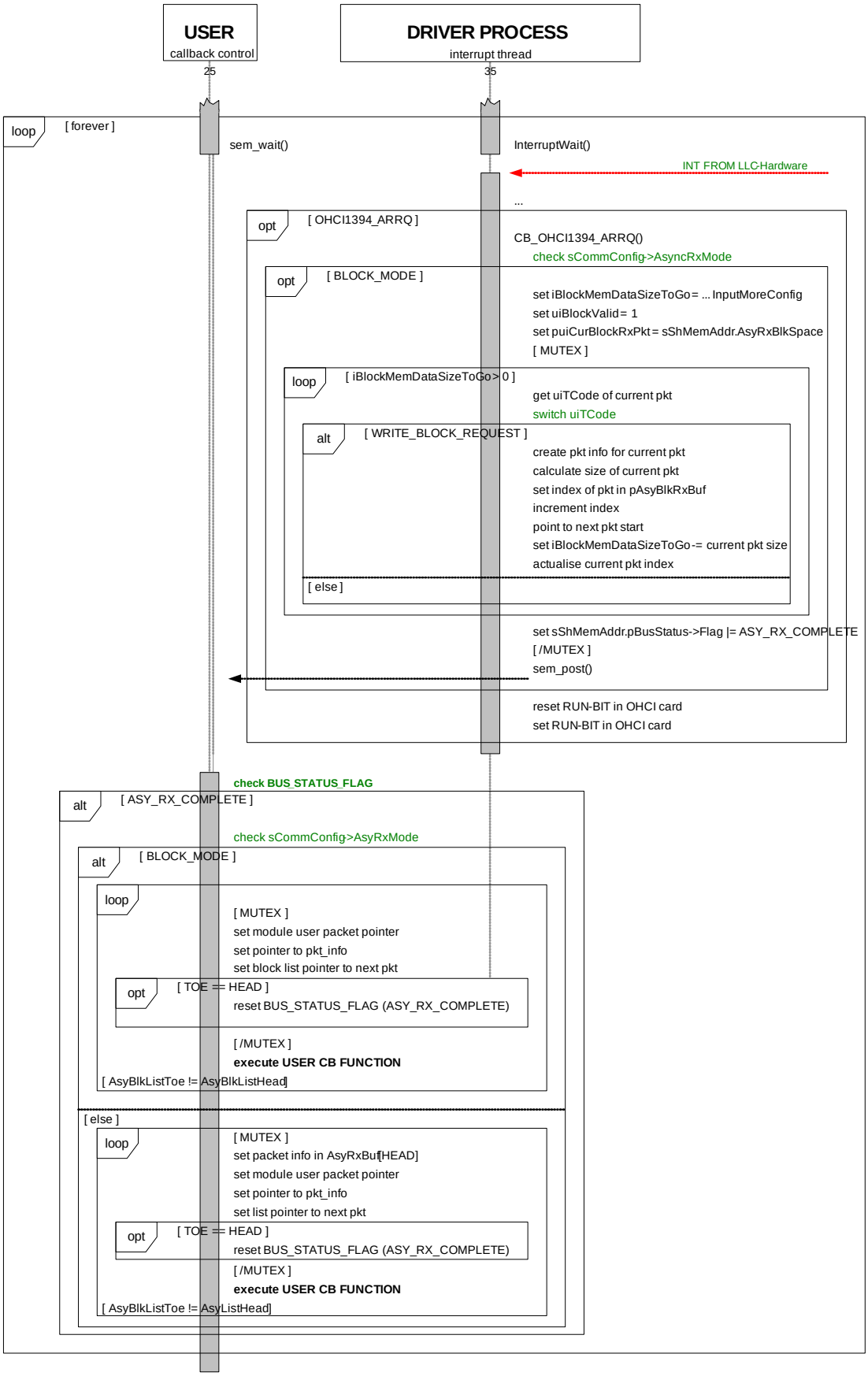



Abbildung 4.54: IEEE1394_SW_QNX_EmgHW - Seq_RxAsyBlk

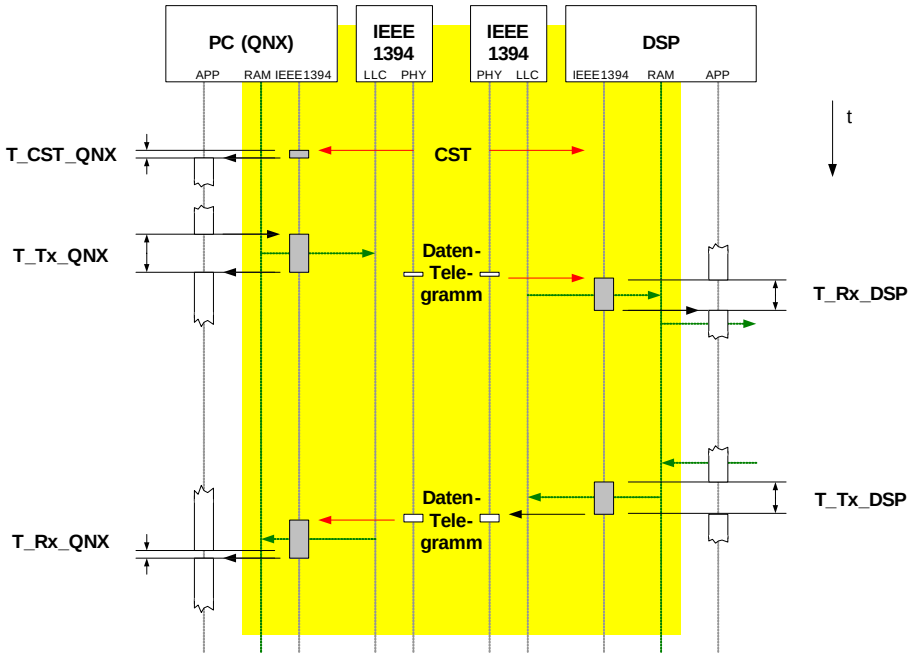



Abbildung 4.55: Timing-Diagramm für Datentransfer über IEEE1394

Das Absetzen der Semaphore bewirkt das Aktivieren des CallbackControlThread innerhalb des Anwendungsprozesses. Dieser prüft nun die BusStatusFlags und setzt die Datenzeiger und weitere Informationsstrukturen des Anwenders gemäß der empfangenen Telegramme, sofern asynchrone Telegramme als Block empfangen worden sind. Hier erfolgt dann auch der Aufruf der zuvor eingerichteten UserCallback-Funktion, die die Auswertung der empfangenen Daten Telegramm für Telegramm erledigt.

### Kommunikationsablauf und Messzeiten

Abbildung 4.55 zeigt das Sequenzdiagramm für einen exemplarischen Kommunikationsvorgang zwischen dem PCI-Treibermodul und einem knotenseitigen Treibermodul. Dargestellt ist der zeitliche Aufwand für einzelne Funktionalitäten innerhalb der Treibersoftware zur Übertragung von Daten. Die Zeitmessung bezieht sich jeweils auf die Zeit zwischen dem Erscheinen eines Telegramms auf dem Bus und dem jeweiligen Beginn/Ende des Schreibvorgangs im allgemeinen, lokal zugänglichen RAM-Speicher.

Der Treiber ist in der Lage auf die verschiedenen Bus-Ereignisse zu reagieren und daraufhin UserCallbacks aufzurufen, die von der übergeordneten Softwareschicht (hier der Kommunikationsprotokoll-Realisierung) mit spezifischen Funktionalitäten belegt sind. So erfolgt auf den Empfang eines jeden CST während der Zeit `T_CST_QNX` zunächst das Erkennen und Auswerten des Ereignisses, gefolgt vom Inkrementieren eines Zyklenzählers, der beim Erreichen eines vom Anwender vorgegebenen Wertes die Ausführung des UserCallbacks bewirkt. Damit lassen sich Zykluszeiten für das Steuerungssystem einfach realisieren:

| Zeitparameter | Zeit in $\mu s$ |
|---|---|
| T_CST_QNX | 3 |
| T_Tx_QNX | 30 |
| T_Rx_DSP | 20 |
| T_Tx_DSP | 20 |
| T_Rx_QNX | 4 (25) |

Tabelle 4.2: Gemessene Ausführungszeiten von IAP-Funktionen

$$AnwenderZykluszeit = n * 125\mu s$$

Der Anwender bekommt also nur jeden $n$-ten Hardwarezyklus des Kommunikationssystems mit. Wird dann vom Anwender ein Sendeauftrag erteilt, so erfolgt innerhalb des Treibers während der Zeit `T_Tx_QNX` eine Beauftragung des DMA-Controllers und das Übertragen der gewünschten Daten aus dem RAM in den LLC. Das Telegramm wird vom PHY des PC auf den Bus geschrieben und gleichzeitig von den PHYs aller angeschlossenen Knoten eingelesen.

Sobald ein Telegramm dort vollständig empfangen ist, erfolgt ein Interrupt an den Treiber, der während der Zeitspanne `T_Rx_DSP` den Rx-Fifo-Speicher des LLC vollständig in das RAM überträgt, Telegramminformationen extrahiert und einen Zeiger auf die Telegrammdaten an die entsprechende UserCallback-Funktion übergibt, die die weitere Verarbeitung der Daten übernimmt. Das Schreiben der Daten auf den Bus geht analog dazu während der Zeit `T_Tx_DSP` vor sich. Dazu wird der gewünschte Datenbereich in das Tx-FIFO des LLC übertragen und von dort automatisch an den PHY, dann den Bus weitergereicht.

Während der Zeit `T_Rx_QNX` erfolgt das Auswerten des bereits über DMA erfolgte Übertragen des Telegramms in den RAM. Die Auswertung beschränkt sich dabei auf organisatorische Funktionalitäten zum Extrahieren von Telegramminformationen und der Koordination von Datenspeicheradressen für den Anwender. Über den Aufruf des entsprechenden UserCallback wird die Auswertung der Daten an den Anwender übertragen.

Die Zahlenwerte zu den erwähnten Zeitvariablen wurden über Messungen ermittelt. Sie sind in Tabelle 4.2 zusammenfassend angegeben. Der Messwert für `T_Rx_QNX` ergibt sich aus dem Umstand, dass es von der Wahl der DMA-Priorität und der momentanen Verwendung des PCI-Datenbusses durch den Anwender abhängt, ob der Anwendungsprozess durch den Empfang des Telegramms für bis zu wenige zehn Mikrosekunden verzögert wird.

### 4.4.3 Zusammenspiel MiRPA-X, IAP und IEEE1394

In Abbildung 4.56 ist das Zusammenspiel der in den Abschnitten 4.2, 4.3 und 4.4 vorgestellten Softwarekomponenten der Kommunikations-Infrastruktur dargestellt.

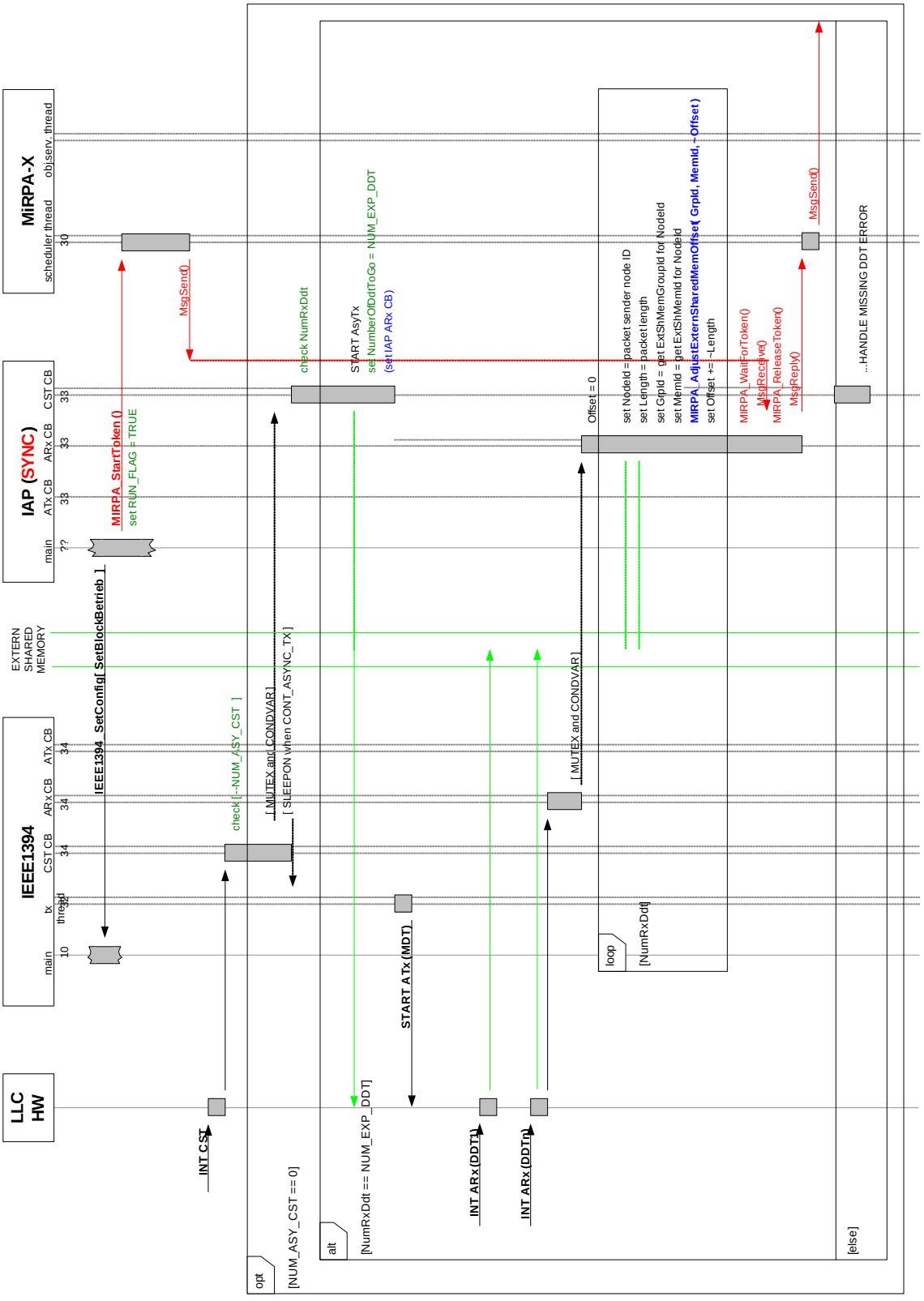



Abbildung 4.56: IAP_QNX_SW_DataHandling.vsd, SEQ - RxBlk

# 5 Anwendung im SFB 562

Der Sonderforschungsbereich (SFB) ist ein von der Deutschen Forschungsgemeinschaft (DFG) gefördertes Forschungsvorhaben, das in Schritten von jeweils drei oder vier Jahren mit einer maximalen zeitlichen Dauer von 12 Jahren gefördert werden kann. Forschungsthemen werden dabei nicht unbedingt von der DFG vorgegeben, sondern sollen sich an den aktuellen oder möglichen zukünftigen Erfordernissen in der Wirtschaft orientieren. Jede Förderungsperiode muss erneut von den bearbeitenden Forschungseinrichtungen beantragt und von der DFG bewilligt werden.

In den folgenden Abschnitten erfolgt eine Erläuterung der Problemstellung für den SFB 562 „Robotersysteme für Handhabung und Montage“ sowie eine Zusammenfassung der Zielsetzung. Ausgehend von der allgemeinen Betrachtung erfolgt anschließend eine Einordnung der vorliegenden Arbeit in das Sonderforschungsprojekt und schließlich eine umfassende Beschreibung des Einsatzes der in dieser Arbeit entwickelten technischen Realisierungen innerhalb des SFB 562 und seiner Demonstratoren.

### Einordnung des SFB 562

Ziel des Sonderforschungsbereiches 562 ist die Erarbeitung von methoden- und komponentenbezogenen Grundlagen für die Entwicklung von Robotersystemen auf der Basis geschlossener kinematischer Ketten (Parallelroboter), um damit das strukturelle Potenzial dieser Roboter besser erschließen zu können. Dabei stehen vor allem hohe erreichbare Geschwindigkeiten und Beschleunigungen im Vordergrund, wie sie im Anwendungsfeld der Handhabung und Montage erforderlich sind (s. Abbildung 5.1) [131].

Neben der Lösung parallelroboterspezifischer Problemstellungen im Bereich der Modellierung und Steuerungstechnik liegt ein weiterer Schwerpunkt der Forschungsarbeiten auf der konsequenten Nutzung der strukturbedingten Vorteile dieser Roboter durch den Einsatz neuer, angepasster Maschinenelemente. Von besonderer Bedeutung ist in diesem Zusammenhang die Adaptronik. Durch die Integration aktiver Komponenten in das Robotersystem und deren gezielte Ansteuerung werden die bereits guten dynamischen Eigenschaften von Parallelstrukturen weiter verbessert. Aufgrund des strukturellen Aufbaus aus massearmen Stäben lässt sich die Adaptronik gerade bei Parallelstrukturen wirtschaftlich einsetzen.

Der Sonderforschungsbereich gliedert sich in folgende Projektbereiche:

**Projektbereich A: Entwurf und Modellierung** befasst sich mit den Grundlagen zum Entwurf und zur Modellierung von Parallelrobotern mit adaptronischen Kompo-

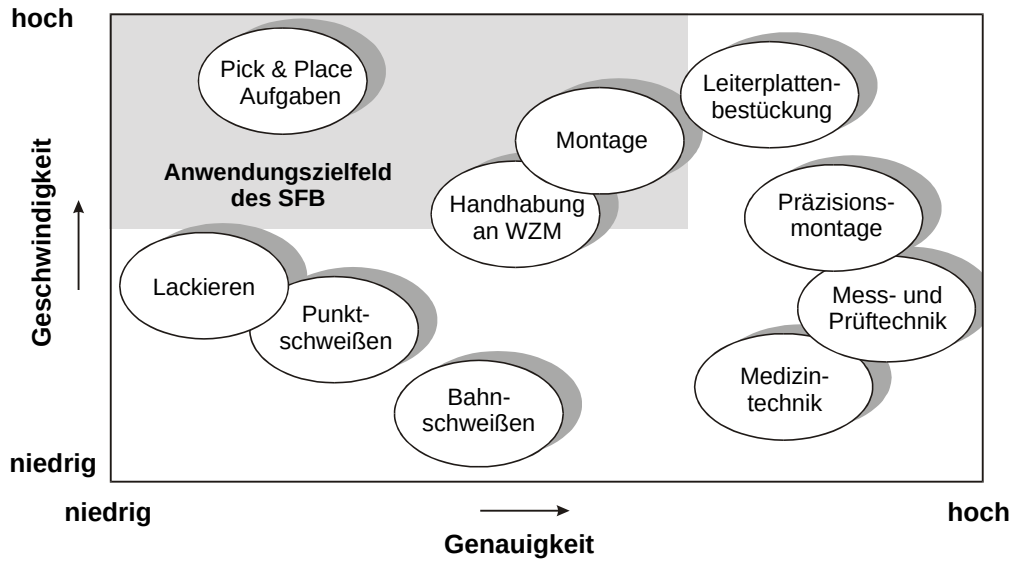



Abbildung 5.1: Einsatzgebiete von Industrierobotern (Quelle: SFB [131])

nenten. Dies erfolgt unter den Randbedingungen von Handhabungs- und Montageanwendungen mit hohen dynamischen Anforderungen und unter Einbeziehung der Potenziale und Restriktionen, die sich aus den anderen Projektbereichen ergeben.

**Projektbereich B: Steuerung und Informationsverarbeitung** befasst sich mit der Erarbeitung neuer parallelroboterspezifischer Lösungen im Bereich der Steuerungstechnik und der Informationsverarbeitung. Dabei wird insbesondere die innerhalb des Projektbereichs C entwickelte Prozessperipherie sowohl unter hardware- als auch unter softwaretechnischen Gesichtspunkten berücksichtigt.

**Projektbereich C: Neue Komponenten** befasst sich mit der Entwicklung und Optimierung von Maschinenkomponenten, die zum einen den speziell bei Parallelrobotern auftretenden Anforderungen gerecht werden können und zum anderen das strukturbedingte Potenzial dieser Roboter hinsichtlich der dynamischen Leistungseigenschaften besser erschließen. Besonders zu nennen ist hier die gelenkintegrierte Sensorik und ihre erforderliche Einbindung in die Datenkommunikation.

Aufgrund der strukturellen Unterschiede sind bei Robotern auf der Basis geschlossener kinematischer Ketten eine Reihe von Besonderheiten zu berücksichtigen, die bei seriellen Robotern so nicht auftreten. Diese führen zu parallelroboterspezifischen Problemstellungen auf den Gebieten der Modellierung, der konstruktiven Umsetzung sowie der Steuerung, Regelung und Informationsverarbeitung. Die in dieser Arbeit beschriebenen Entwicklungen aus dem Abschnitt 4 beziehen sich dabei vor allem auf das zuletzt erwähnte Gebiet der Steuerung, Regelung und Informationsverarbeitung, dessen Zielsetzung im folgenden Absatz beschrieben wird. Sinn dieser Beschreibung ist die Darstellung thematischer Schnittstellen mit beteiligten Teilprojekten und da-

mit einer ersten übersichtsartigen Anforderungsbeschreibung für die in dieser Arbeit entwickelte Hard- und Software.

### Zielsetzung für Projektbereich B

Zielsetzung des SFB 562 im Projektbereich für Regelung, Steuerung und Informationsverarbeitung ist es, ein leistungsfähiges Steuerungssystem aufzubauen und zu betreiben, an dem Teile der zeitkritschen Anwendungen (Komponenten) über ein schnelles Kommunikationssystem angebunden werden. Die Formulierung eines Anwendungsprofils für parallelkinematische Prozesseinheiten, das die Anwendungs- und Kommunikationsebene sowie die Anwenderdaten in Bezug auf Interoperabilität umfasst, soll nicht nur verschiedene Dienste und Funktionen zur Kommunikation enthalten, sondern beispielsweise auch Vorgaben, welche Geräte und Sensoren für die ausgewählte Art der Applikation notwendig sind. Es müssen neue Kommunikationsprofile und -protokolle entwickelt werden, da der Einsatz von zusätzlich integrierter Sensorik sowie der Adaptronik neue Anforderungen bezüglich Übertragungsgeschwindigkeit und Synchronisation stellt.

Um eine systematische und effiziente Softwareentwicklung und Softwareintegration in eine Steuerung für Parallelroboter zu ermöglichen, sollen verschiedene Softwaretechnologien, wie beispielsweise Modularisierungskonzepte, eingesetzt werden. Die so entwickelten Softwaremodule sollen in ein sogenanntes Baukastensystem integriert werden. Dabei sollen auch bestehende Steuerungs- und Kommunikations-Standards untersucht und unter parallelroboterspezifischen Gesichtspunkten erweitert werden. Aufbauend auf dem objektorientierten Ansatz wird eine in austauschbare Komponenten aufgegliederte Struktur angestrebt. Da sich die Komponenten nicht nur über die zu definierenden Schnittstellen beeinflussen, sondern auch implizit durch den Zugriff auf gemeinsame Rechnerressourcen, berücksichtigt ein komponentenhafter Ansatz neben strukturellen auch dynamische Aspekte. Insbesondere wird durch Verhaltensmodellierung der Komponenten sichergestellt, dass deren Echtzeitanforderungen im Gesamtsystem nicht verletzt werden. In dem sich daraus ergebenden Verhaltensmodell des Gesamtsystems kann dann die Konsistenz zu den Echtzeitbedingungen der einzelnen Komponenten überprüft werden. Vorteil dieses Vorgehens ist, dass sich dynamische Aspekte schon frühzeitig auf der Spezifikationsebene berücksichtigen lassen.

Im Bereich der Regelungstechnik werden neue Regelkonzepte unter Berücksichtigung positions- und richtungsabhängiger Robotereigenschaften entwickelt, die insbesondere den großen Einfluss der variablen Last auf die Gesamtträgheit des Parallelroboters berücksichtigen. Die zusätzliche interne Sensorik sowie die adaptronischen Komponenten sollen dabei während der Reglersynthese explizit berücksichtigt werden. So soll z.B. die strukturintegrierte Kraftsensorik neben der Messung von Kontaktkräften auch zur Bestimmung der Nutzlast verwendet werden, um eine bestmögliche Regleranpassung und dynamische Entkopplung der kraft- und lagegeregelten Koordinatenachsen zu gewährleisten.

Des Weiteren werden im SFB 562 neue maschinennahe Steuerungsfunktionen auf ihren Einsatz an Parallelrobotern hin entworfen und unter steuerungs- und softwaretechni-

schen Aspekten untersucht und optimiert. Auch hier sollen die zusätzlich integrierten Sensoren verwendet werden. Die der Steuerung zur Verfügung stehenden redundanten Informationen interner Sensoren des Parallelroboters sollen zur analytischen Lösung des DKP oder zur Entwicklung einer rechenzeiteffizienten Arbeitsraumüberwachung verwendet werden. In diesem Zusammenhang wird auch die Frage nach der optimalen Sensorkonfiguration sowie der notwendigen Sensorauflösung behandelt.

Die Zielsetzung des SFB 562 verlangt die synergetische Zusammenführung fachübergreifenden Wissens aus den Bereichen Maschinenbau, Elektrotechnik und Informatik. Diese Forderung wird über die Ausrichtung der beteiligten Institute erfüllt. Die Ausrichtungen sind dabei im einzelnen:

- Automatisierungs- und Handhabungstechnik
- Robotik
- Strukturmechanik / Adaptronik
- Konstruktionstechnik / Maschinenelemente
- Steuerungs- und Regelungstechnik
- Softwareentwicklung
- Kommunikationstechnik
- Antriebstechnik
- Mikrotechnik / Sensorik

Der folgende Abschnitt beschreibt nun den spezifischen Aufgabenbereich, in dem die in dieser Arbeit entwickelten Applikationen zum Einsatz kommen.

#### Zielsetzung für Teilprojekt B1

Roboter, die konstruktiv auf geschlossenen (parallelen) kinematischen Strukturen aufgebaut sind, weisen steuerungstechnische Probleme auf, wie sie bei seriellen Strukturen nicht vorkommen, insbesondere, wenn konstruktive Elemente aktiv, d.h. adaptronisch, ausgeführt sind. Eine zentrale Steuerung mit Berechnung der Trajektorie, Arbeitsraumüberwachung und Konfliktstrategien sowie der Regelung der Antriebe auf der Momentenebene bietet den Vorteil eines schnellen Austausches von Informationen unter den verschiedenen Programmmodulen und gewährleistet somit ein hochdynamisches Verfahren der Struktur. Dieser Ansatz bedingt aber gleichzeitig eine zeitkritische Verteilung verschiedener Informationen zwischen der Steuerung und externen Komponenten (Antriebe, adaptronische Elemente, Sensoren, etc.) im Stromreglertakt. Kommunikationssysteme, wie sie sich im Aktor- und Sensorbereich räumlich ausgedehnter Anlagen bereits etabliert haben [150], sind für diese Anforderungen nicht geeignet. Die erforderliche Bandbreite liegt um eine Zehnerpotenz über den

Abbildung 5.2: Demonstrator „HEXA“ des SFB 562

aktuell angebotenen Systemen, zudem müssen Mechanismen zur hochgenauen Synchronisation der Teilnehmer bereitgestellt werden. Die Entwicklung eines geeigneten Kommunikationssystems ist eine Aufgabe dieses Teilprojekts.

Die folgenden Abschnitte beschreiben die Architektur und die Hard- und Software-Struktur der im SFB 562 realisierten Steuerung für den „HEXA“-Roboter (Abbildung 5.2) sowie den Einsatz der Kommunikations-Infrastruktur innerhalb dieser Steuerung.

## 5.1 Realisierung der Steuerung

Ein zentrales Anliegen des SFB 562 bei der Entwicklung von Komponenten für Robotersysteme ist die Erhöhung der Positionier- und Orientierungs-Wiederholgenauigkeit eines Greifers trotz gesteigerter Platziergeschwindigkeit im Vergleich zu bekannten Robotersystemen. Erschwerend bei der Positionierung und Orientierung kommt hinzu, dass auch die Fertigungsgenauigkeiten von zu greifenden Bauteilen oder auch die angenommene Position der Bauteile auf der Arbeitsplattform Schwankungen unterliegen können, die durch die Steuerung ausgeglichen werden müssen. Hinsichtlich dieser Fehlertoleranz können dabei entscheidende Verbesserungen durch die Verwendung des „Task Frame Formalismus“ [92] [96] zur Beschreibung der spezifischen Montageprozesse im Zusammenhang mit der Realisierung einer hybriden Regelung, das ist die Berücksichtigung von Sensorinformationen zur Regelung jedes einzelnen Roboter-Freiheitsgrades mit einem separaten Regelalgorithmus, erzielt werden. Eine signifikante Steigerung der Wiederholgenauigkeiten hinsichtlich der Positionierung und Orientierung und eine leistungsverbesserte sensorgeführte Bewegung kann durch den Einsatz von Parallelkinematiken erzielt werden, da sie über eine größere Strukturseifigkeit und geringere Strukturträgheiten verfügen als serielle Kinematiken. Es wird

davon ausgegangen, dass das „Aktionsprimitiv“-Denkmuster bei der Programmierung von Bewegungsfunktionen (s. Abschnitt 5.1.1) eine natürlichere Sicht auf die Roboteraufgabe unter Nutzung von hybrider Regelung bei der Beschreibung mit Hilfe des Task Frame Formalismus ermöglicht. Letzterer lässt sich gut in die Aktions-Primitiv-Struktur einbeziehen. Außerdem lassen sich Aktionsprimitive nach der Spezifikation von symbolisch-räumlichen Relationen auch automatisch aus CAD-Daten der Montageteile generieren [97].

### 5.1.1 Bewegungsprogrammierung mit Aktionsprimitiven

Die Programmierschnittstelle der Aktionsprimitive erlaubt es dem Programmierer der Roboteraufgabe, die beabsichtigte Endeffektorbewegung bezüglich eines frei wählbaren kartesischen Koordinatensystems (s. Task Frame Formalismus, Seite 178) anzugeben. Für jeden Freiheitsgrad, der über dieses Koordinatensystem abgedeckt wird, kann innerhalb des Aktionsprimitives ein eigener Bewegungs-Regelalgorithmus spezifiziert werden. Diese Freiheit bei der Bewegungsplanung führt auf der einen Seite zu einer intuitiven Beschreibungsmöglichkeit für die hybride Regelung innerhalb der Bewegungsaufgabe und ermöglicht das Zusammenfügen von Netzen (Aktionsprimitiv-Netzen) zur Beschreibung ganzer Montagevorgänge. Auf der anderen Seite führt diese intuitive Beschreibungsmöglichkeit zu besonderen Anforderungen an die zu realisierende Architektur der Steuerung. Eine besondere Anforderung ist dabei die Notwendigkeit, zu jedem Zeitpunkt innerhalb eines Regelungszyklus die momentan ablaufende Kombination aus Regelungsalgorithmen für die verschiedenen Freiheitsgrade zu verlassen und durch neue zu ersetzen, sollte die sensorgeführte Bewegungsaufgabe dies erfordern. Dieses z.B. wird immer dann erforderlich, wenn eine Stoppbedingung (ebenfalls im Aktionsprimitiv spezifiziert) erfüllt ist. Wenn diese Anforderung berücksichtigt wird, muss wiederum jeder einzelne Bewegungsalgorithmus in der Lage sein, Geschwindigkeits- und Beschleunigungs-Initialwerte zu verarbeiten, die ungleich Null sind. Das wiederum führt zu der Forderung nach der Verwendung von echtzeitfähigen Bewegungsplanungs-Algorithmen, die „online“ berechnet werden, und damit nach großer Kommunikationsbandbreite.

#### Task Frame Formalismus

Der Unterschied des Task Frame (TF) Formalismus zu konventionellen Bewegungsbeschreibungen liegt darin, dass die Bewegung immer im kartesischen Task Frame definiert ist; dieser Task Frame selbst kann aber frei innerhalb des Roboterkoordinatensystems platziert werden. Eine Ankerbeziehung definiert die Bewegung des Endeffektors, welcher über den Hand Frame (HF) repräsentiert ist (Abbildung 5.3).

Die Definition von Ankerbeziehungen ist eine wichtige Voraussetzung zur Realisierung von „nachgiebiger“ (sensorgeführter) Bewegung und der Ausführung von geometrisch fehlertoleranten Montagesequenzen (s. Beispiel, Seite 179). Wenn beispielsweise der TF um seine z-Achse rotiert (s. Abbildung 5.3), wobei der HandFrame mit dem TF verankert ist, dann folgt der HF unter der Ausführung von Rotations- und Translationsbewegungen einer zirkulären Bahn, falls der HF und der TF nicht kongruent

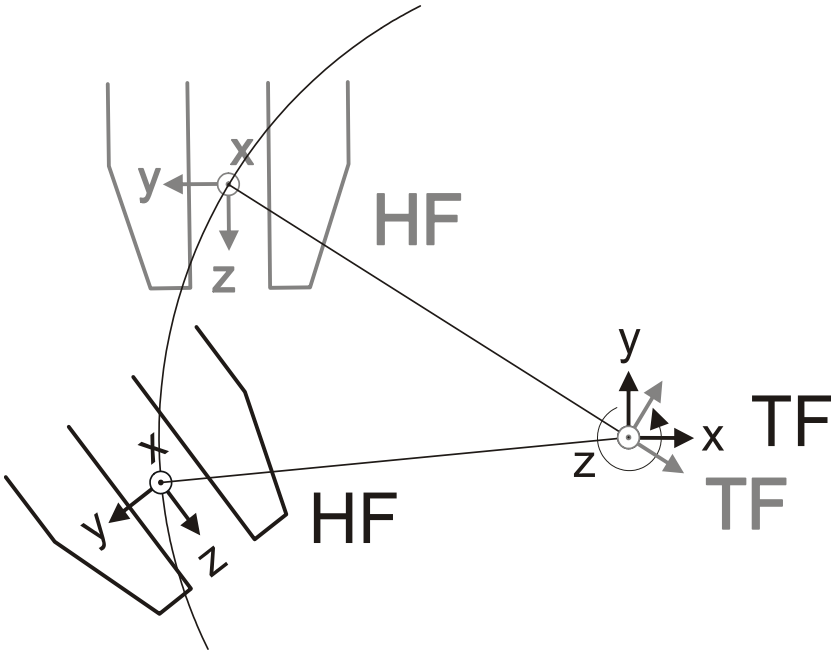



Abbildung 5.3: Mit Task Frame (TF) verankerter Hand Frame (HF)

sind. Wie schon erwähnt, erlaubt die Aktionsprimitiv-Schnittstelle dem Bewegungsprogrammierer die separate Definition von Bewegungsalgorithmen für jeden der verfügbaren Freiheitsgrade des TF. Damit werden hybride (sensorgeführte) Bewegungen des HF ermöglicht. Sollte ein Regelungsalgorithmus nicht in der Lage sein, seine Regelaufgabe für den spezifizierten Freiheitsgrad zu erfüllen, wird er durch einen vom Programmierer bestimmten Alternativregler (im Aktionsprimitiv festgelegt) ersetzt. Beispielsweise soll die y-Achse des TF kraftgeregelt werden; allerdings befindet sich der Endeffektor noch nicht in einer Kontaktsituation. Also würde hier z.B. ein Geschwindigkeitsregler den Kraftregler ersetzen, bis der Kontakt eingetreten ist.

#### Beispiel einer fehlertoleranten Montage

Um die Vorteile bei der Bewegungsprogrammierung mit Hilfe von Aktionsprimitiven zu zeigen, wird nun ein auf zwei Dimensionen reduziertes Montagebeispiel erläutert. Die Aufgabe besteht darin, einen Quader in ein dafür vorgesehenes Loch zu stecken. Die dafür notwendigen Arbeitsschritte sind in Abbildung 5.4 dargestellt. Das dargestellte Koordinatensystem ist der TF.

Zunächst wird geschwindigkeitsgeregelt in y-Richtung des TF verfahren, bis sich ein Kontakt mit der Oberfläche der Arbeitsplattform einstellt (a). Bei Kontakt wird nun auf eine Kraft in y-Richtung geregelt und mit dem resultierenden Moment um die z-Achse der x-Abstand des TF vom Kontaktpunkt berechnet (b). Genauso erfolgt die Berechnung des y-Abstands vom Kontaktpunkt über die Applikation einer Kraft in negative x-Richtung (c). Mit diesen Angaben kann nun der TF in den Kontaktpunkt verschoben werden (d); eine Rotation um die z-Achse ergibt den Kontakt der Oberflächen (detektiert über ein gemessenes Drehmoment um die z-Achse). Nun ist

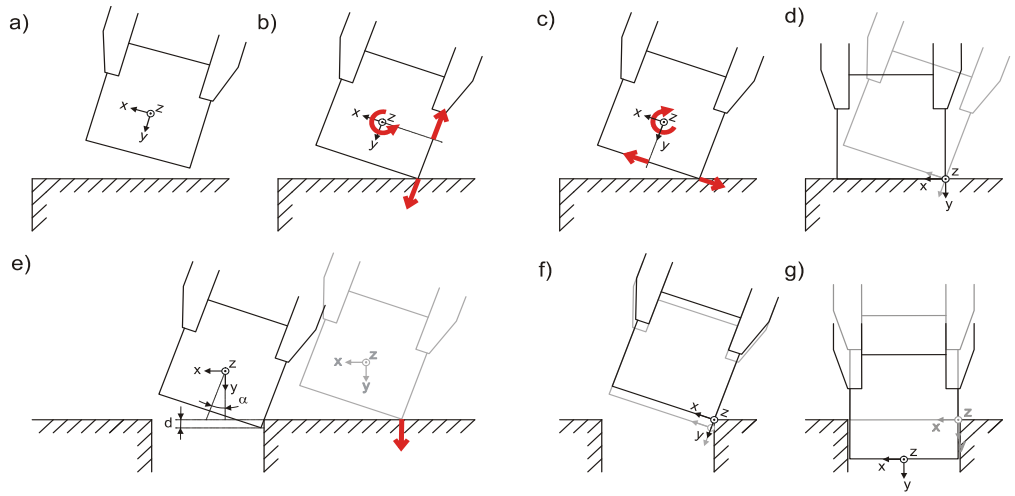



Abbildung 5.4: Beispiel - Montage unter Nutzung von Aktions-Primitiven

die Position des Quaders relativ zum Greifer bekannt; der TF wird in die Mitte des Quaders verschoben und der Quader selbst mit einer geregelten Andruckkraft in y-Richtung entlang der x-Achse so lange verschoben, bis ein Versatz $d$ in y-Richtung gemessen wird (e). Über den Cosinus von $\alpha$ und $d$ ist so der Abstand zwischen der Ecke des Quaders und des Loches bekannt. Der TF wird in die Ecke des Quaders gelegt, um $\alpha$ in negative z-Richtung gedreht und anschließend der Quader um den Abstand in negative y-Richtung verschoben (f). Nun erfolgt eine Drehung um $\alpha$ in z-Richtung, so dass der Quader über dem Loch steht, und ein Verschieben des Quaders ins Loch (g). Obwohl es zu Anfang der Montageaufgabe eine Reihe von unbestimmten Toleranzen gab, stellte sich diese Abfolge von Aktions-Primitiven als eine verlässliche Montagestrategie heraus.

### 5.1.2 Eine universelle Robotersteuerung

In diesem Abschnitt wird der generelle Aufbau der realisierten Robotersteuerung erläutert. Zuerst wird aus regelungstechnischer Sicht ein Systementwurf vorgestellt, der das Umsetzen des Task Frame Formalismus und die Integration frei wählbarer Sensoren zur Realisierung von hybriden Regelungsalgorithmen ermöglicht. Die Stärken dieser Struktur sind seine Modularität und die Fähigkeit, hybride Regelungsstrategien zur Laufzeit gemäß des aktuellen Aktionsprimitives zu modifizieren. Anschließend werden aus softwaretechnischer Sicht die Kommunikationstechniken und die Entwurfsmethodik für die Robotersteuerung beschrieben.

#### Aspekte aus der Steuerungstechnik

Abbildung 5.5 stellt die Sicht auf die Steuerungsarchitektur zur Ausführung von Aktionsprimitiven dar. Als Kernkomponenten sind die Bewegungsmodule (motion mo-

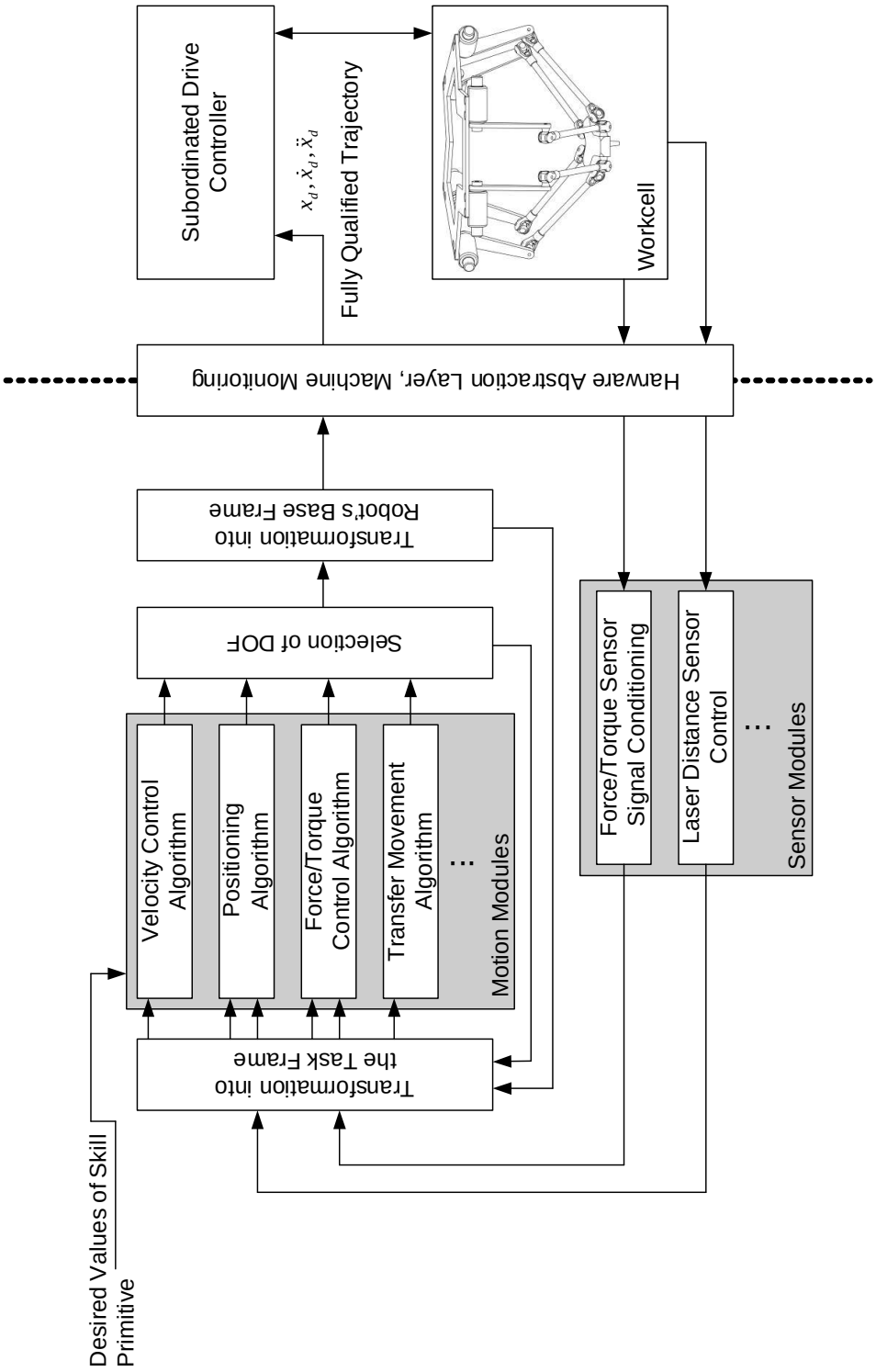



Abbildung 5.5: Struktur der modularen Steuerung

dules) zu nennen, die in der Mitte der Abbildung dargestellt sind. Um eine Trajektorie des Endeffektors zu berechnen, die auf Entfernungs-, Kraft/Momenten- oder Geschwindigkeitsregelung beruht, verwenden die Bewegungsmodule entweder gemessene Sensordaten aus dem Feld oder kommen ohne diese aus. Die Bewegungsalgorithmen arbeiten alle im TF und behandeln jeden der Freiheitsgrade als voneinander unabhängig. Diese Struktur erlaubt das automatische Erzeugen von Montageabläufen, birgt aber gleichzeitig Stabilitätsproblematiken in sich, auf die in [97] eingegangen wird. Jedenfalls konnten diese kritischen Sachverhalte gelöst und in ersten Implementierungen zufriedenstellend realisiert werden [98]. Der Roboter selbst ist über einen untergeordneten, roboterspezifischen Antriebsregler (subordinate drive controller) geregelt, der über die Eingangsgrößen Position, Geschwindigkeit und Beschleunigung in Roboterkoordinaten eine geschlossene Regelung realisiert. Dieses ermöglicht es dem Entwickler, moderne, modellbasierte Regelungsalgorithmen wie den bekannten „computed torque“[1]-Algorithmus zur Anwendung zu bringen, der zur Regelung von Parallelkinematiken erforderlich ist. Der untergeordnete Antriebsregler muss mit Vorsicht realisiert werden, um stabile Regelzustände über den gesamten Roboterarbeitsraum mit passend definierten dynamischen Randbedingungen (Maximalwerte für Ruck, Beschleunigung und Geschwindigkeit) sicherzustellen. Folglich wurde der Kraft/Momentenregler als Impedanzregler implementiert, so wie es in [99] vorgeschlagen und experimentell verifiziert wurde. Im folgenden Absatz wird ein exemplarischer Steuerungszyklus beschrieben.

Bei jedem Empfang eines neuen Aktionsprimitives wird ein Satz relevanter Daten an die Bewegungsmodule übertragen. Dazu werden die aktuelle Position, Geschwindigkeit und Beschleunigung in den TF des aktuellen Aktionsprimitivs transformiert, um den Bewegungsmodulen die Berechnung von ruckbegrenzten Trajektorien zu ermöglichen. Nun kann das Aktionsprimitiv in der Steuerung ausgeführt werden. In jedem Steuerungszyklus werden die Ausgangsdaten zu einem einzelnen Ausgangsvektor, der relativ zum TF die gewünschte Position, Geschwindigkeit und Beschleunigung enthält, zusammengefasst. Diese Zusammenstellung ist abhängig davon, ob die Bewegungsalgorithmen ein entsprechendes Ergebnis berechnen konnten oder nicht. Wenn z.B. der Endeffektor keinen Kontakt mit einem Objekt oder der Arbeitsplattform hat, ist der Kraftregler nicht in der Lage, eine gewünschte Trajektorie für die entsprechende Koordinatenachse zu berechnen. Dann wird ein solches unpassendes Ergebnis durch das Ergebnis eines vom Programmierer spezifizierten Regelungsalgorithmus einer unteren Ebene (Alternativregler) ersetzt (z.B. eines Geschwindigkeitsreglers). In dem folgenden Schritt wird die aktuelle Trajektorieninformation (Ausgangsvektor relativ zum TF) für den nächsten Steuerungszyklus wieder zurück in die Bewegungsmodule geführt. Dieses ist notwendig, damit kurzzeitig deaktivierte Regelungsalgorithmen (z.B. der Kraftregler) mit stets aktuellen Daten betrieben werden können. Schließlich wird der Ausgangsvektor vom TF in das Basiskoordinatensystem des Roboters transformiert, um damit die Sollwerte für den unterlagerten Antriebsregler bereitzustellen. Danach ist die Steuerung bereit für den nächsten Steuerungszyklus oder ein nächstes Aktionsprimitiv.

[1] Algorithmus zur Berechnung der Drehmomentvorsteuerung aus dem inversen dynamischen Modell des Parallelroboters

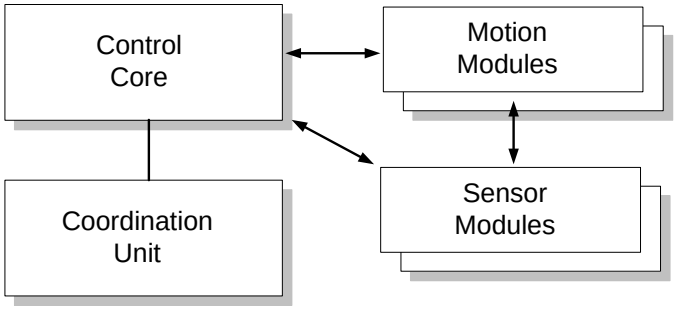



Abbildung 5.6: Modularer Trajektoriengenerator für den Task Frame Formalismus

#### Aspekte aus der Softwaretechnik

Der roboterunabhängige Trajektoriengenerator ist das zentrale Element der Steuerungsarchitektur und besteht aus den Funktionsmodulen, die in Abbildung 5.6 dargestellt sind.

Der Steuerungskern (control core) interagiert, unterstützt durch die Koordinationseinheit (coordination unit), mit der Aktionsprimitiv-Programmierschnittstelle und bildet auf dynamische Weise Kommunikationspfade zwischen dem Kern, den Bewegungsmodulen und Sensormodulen, je nach der durch das aktuelle Aktionsprimitiv spezifizierten Bewegungsaufgabe. Diese dynamische An- und Abkopplung von Modulen wird durch die Nutzung der Middleware MiRPA-X ermöglicht, die speziell unter Berücksichtigung der Leisungsanforderungen innerhalb von hochperformanten Roboter- und Bewegungsanwendungen entwickelt wurde. Die Middleware und ihre Eigenschaften sind in Abschnitt 4.2 im Rahmen der Beschreibung der Kommunikations-Infrastruktur aus Abschnitt 4 genauer beschrieben, während in dem folgenden Abschnitt spezieller auf die Anwendung innerhalb des im SFB 562 entwickelten Robotersystems eingegangen wird.

Beim Start der Steuerung verschickt der Steuerungskern eine Sammel-`COMMAND`-Nachricht an alle im System vorhandenen Bewegungsmodule. Diese beantworten die Anfrage jeweils mit dem Ablegen einer Identifikation und einer zugehörigen Spezifikation ihrer funktionalen Eigenschaften (Möglichkeiten) innerhalb eines vorher festgelegten und gesicherten Speicherbereichs (*shared memory*). Die entscheidenden Eigenschaften sind dabei ihre Einsatzfähigkeit innerhalb eines hybriden Regelungsumfeldes und ihre Unterstützung des Task-Frame-Formalismus für die jeweilige Aufgabe. Mit dieser Information über vorhandene und aktive Bewegungsmodule kann nun der Steuerungskern entscheiden, inwiefern die im aktuellen Aktionsprimitiv gewünschte Modulzusammenstellung zur Laufzeit ausführbar ist und welche Transformationen bei der Ausführung berechnet werden müssen. Die wesentlichen Daten der Bewegungsmodul-Schnittstelle sind links in Abbildung 5.7 dargestellt. Die Istwerte (jeweils 6-dimensionale Vektoren) für Position, Geschwindigkeit und Beschleunigung des Greifers (Hand-Frame, HF) werden hierüber zum entsprechenden Bewegungsmodul übertragen. Diesen Daten folgt eine Liste von zuvor beantragten Shared-Memory-Adressen, an denen das Modul benötigte Sensordaten lesen kann. Zusätzlich wird der Sollwert für den Regelungsalgorithmus und ein Datenfeld zur Auswahl der beabsich-

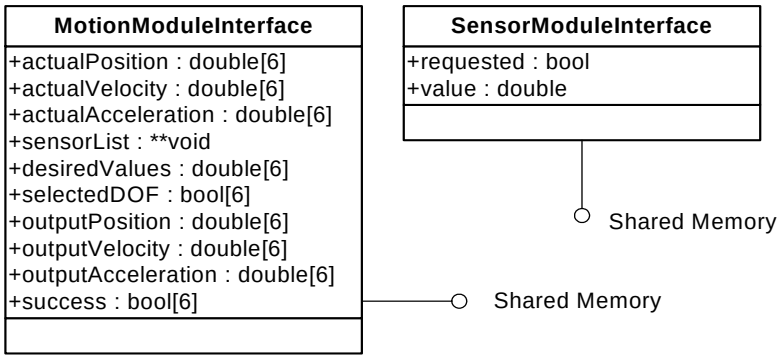



Abbildung 5.7: Modulkommunikation über Shared-Memory-Schnittstellen

tigten Freiheitsgrade im TF übertragen. Der Algorithmus eines Bewegungsmoduls gibt über diese Schnittstelle ein Datenfeld zurück, in dem vermerkt ist, für welche Freiheitsgrade die Ausgangsdaten erfolgreich berechnet werden konnten. Die Ausgangsdaten für Position, Geschwindigkeit und Beschleunigung werden ebenfalls über diese Struktur zurückgegeben.

Die in jedem Trajektoriengenerator-Zyklus zur Ausführung des jeweiligen Aktionsprimitives erforderlichen Bewegungsmodule werden einzeln über jeweils eine `COMMAND`-Nachricht aktiviert. Am Ende jedes dieser Zyklen entscheidet der Steuerungskern anhand des Datenfeldes für die erfolgreich bearbeiteten Freiheitsgrade eines jeden Bewegungsmoduls, welche der Ausgangsdaten in das Basiskoordinatensystem des Roboters transformiert werden. Falls die Bewegungsmodule nicht in der Lage waren, das Freiheitsgrad-Datenfeld erfolgreich zu füllen, wird der Kern in einen Fehlerzustand versetzt und eine automatische Abbruchtrajektorie verwendet, um den Roboter in einen sicheren Bewegungs- oder Ruhezustand zu überführen.

Die Schnittstelle zu den Sensormodulen ist einfacher. Ein Sensormodul legt eines oder mehrere der rechts in Abbildung 5.7 dargestellten Datenstrukturen in einem bei MiRPA-X registrierten Shared-Memory-Bereich ab. Diese Struktur enthält einen Merker, der zunächst vom Steuerungskern gesetzt wird und anzeigt, ob ein Messwert vom Sensormodul zur Berechnung von Ausgangswerten benötigt wird. Aktiviert werden alle Sensormodule dann über das Versenden einer Sammel-`COMMAND`-Nachricht am Anfang eines Trajektoriengenerator-Zyklus. Die beim Empfang einer solchen Nachricht aufgeweckten Sensormodule verarbeiten alsdann Signale oder kopieren einfach Daten von Hardwareschnittstellen in die oben erwähnte Datenstruktur. Vor jedem erneuten Blockieren eines Sensormoduls auf den Nachrichtenempfang im nächsten Zyklus wird jeweils eine MiRPA-X-weit vereinbarte *semaphore* dekrementiert, die zu Anfang eines jeden Zyklus vom Steuerungskern gesetzt wird. Der Initialwert ist dabei die Anzahl an von den Bewegungsmodulen benötigten Sensorwerten für den aktuellen Zyklus. Sobald die Semaphore Null wird, aktiviert der Steuerungskern die Bewegungsmodule. Um ein zeitlich deterministisches Verhalten der Steuerung zu gewährleisten und den Ausfall von Bewegungs- und Sensormodulen abzupuffern, benötigt der Steuerungskern den Prozessor vorrangig vor anderen Modulen zu vorgegebenen Zeiten innerhalb des Zyklus. Dieses wird erreicht, indem vom QNX-Prioritätsvererbungskonzept Ge-

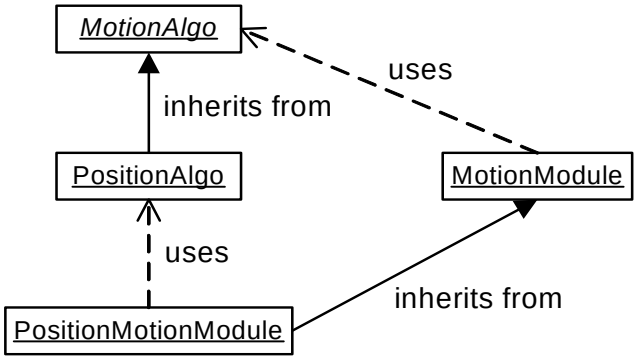



Abbildung 5.8: Kollaborationsdiagramm für ein Bewegungsmodul

brauch gemacht wird (s. Abschnitt 5.1.2). Alle verwendeten SharedMemory-Bereiche und Nachrichten werden über den Namensdienst von MiRPA-X verwaltet.

### Rahmenbedingungen für Bewegungsmodule

Alle Algorithmen zu Robotersteuerung und Bahnplanung sind innerhalb der einzelnen Bewegungsmodule implementiert. Um dabei die Details der Kommunikation und Synchronisation vor den Anwendungsentwicklern des jeweiligen Moduls zu verbergen, werden Rahmenbedingungen und eine allgemeine Rahmenstruktur zur Entwicklung von Bewegungsmodulen festgelegt. Realisiert wird dies über eine abstrakte Klasse (C++), die in Abbildung 5.8 dargestellt ist. Diese Abbildung zeigt das Kollaborationsdiagramm eines Positionsplaner-Bewegungsmoduls. Ein Bewegungsmodul wird in zwei Schritten entwickelt: Im ersten Schritt implementiert der Programmierer seinen Algorithmus, indem er die abstrakte Klasse „MotionAlgo“ erbt und die dort definierten Funktionen mit seinen eigenen Funktionalitäten ersetzt. Im zweiten Schritt erbt er sein Bewegungsmodul von der Basisklasse „MotionModule“, trägt die Eigenschaften seines Algorithmus in der von der Basisklasse vorgehaltenen Profil-Datenstruktur ein und erzeugt eine Instanz des Algorithmus, dessen Objektzeiger an die „MotionModule“-Klasse übergeben wird. Alles Weitere ist in der Basisklasse gekapselt, die die Schnittstelle zum Algorithmus bedient.

### Echtzeit Scheduling für die Steuerung

Das Scheduling der Steuerungsarchitektur ist in drei Ebenen organisiert, wie in Abbildung 5.9 dargestellt. Die Blöcke stellen dabei unterschiedliche Prozesse dar, die auf dem Rechnersystem ablaufen und miteinander über die Mechanismen von MiRPA-X kommunizieren und gegeneinander synchronisiert werden können.

Die unterste Schedulingebene sorgt mit der höchsten Schedulingpriorität für eine harte Echtzeitumgebung, indem hier der von MiRPA-X bereitgestellte Token-Passing-Mechanismus verwendet wird. Die dort angesiedelten Prozesse „field bus communi-

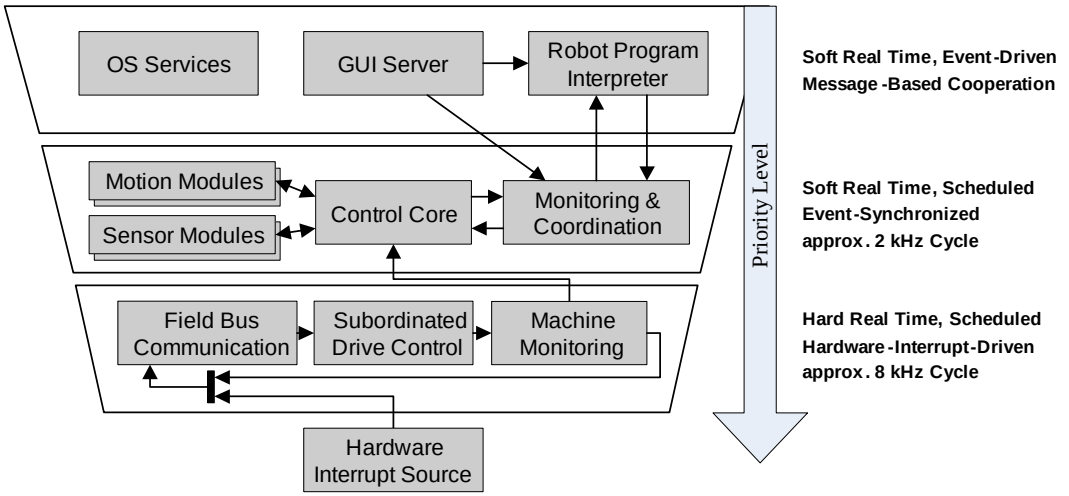



Abbildung 5.9: 3-Ebenen-Schedulingkonzept der Steuerungsarchitektur

cation“, „subordinate drive control“ und „machine monitoring“ blockieren zunächst auf den Empfang eines Tokens von MiRPA-X. Die Verteilung des Tokens an die Prozesse erfolgt zyklisch, sobald ein sogenanntes „StartToken“-Kommando im System abgesetzt wird. Zunächst erhält also der Prozess „field bus communication“, der in Form des Industrial Automation Protocol (IAP, s. Abschnitt 4.3) implementiert wurde, den Token. Dieser versendet Sollwerte für die Antriebe und behält den Token so lange, bis alle Datentelegramme externer Roboterkomponenten (im Allgemeinen sind das die Istwerte der Sensoren) als Antworten eingetroffen sind und verteilt diese anschließend über Mechanismen von MiRPA-X in dafür vorgesehene, über einen eindeutigen Namen referenzierte Speicherbereiche. Anschließend wird der Token freigegeben und gelangt zum untergeordneten Antriebsregler („subordinate drive control“ , s. Abschnitt 5.1.2). Hier werden nun diverse maschinennahe Regelungsfunktionalitäten auf die empfangenen Istwerte angewendet und neue Sollwerte für die Antriebe und andere Aktuatoren berechnet. Von hier aus gelangt der erneut freigegebene Token zum „machine monitoring“-Prozess. Über einen frei einstellbaren, auf die Anzahl empfangener Hardwareinterrupts bezogener Skalierungsfaktor wird entschieden, welche Zykluszeit für die übergeordneten Steuerungsebenen gelten soll. Solange nicht die vorgesehene Anzahl Hardwareinterrupts empfangen wurde, wird einfach der Token wieder freigegeben und gelangt erneut zum ersten Prozess „field bus communication“ der Sequenz. Hier werden zunächst die berechneten Sollwerte auf den Kommunikationsbus geschrieben (zu den Antrieben und Aktuatoren übertragen) und anschließend der Token wieder so lange blockiert, bis alle Sensordaten über den Bus empfangen wurden.

Dieses Verfahren, eine externe Interruptquelle direkt zur Generierung eines Zyklustaktes für Softwarefunktionen zu verwenden, führt zu extrem geringem Jitter (s. Abschnitt 5.1.4) in der Phase und der Ausführungszeit von harten Echtzeitprozessen. Dadurch profitiert der untergeordnete Antriebsregler in signifikanter Weise im Vergleich zur Verwendung anderer Middlewaresysteme (s. OSACA, RT-CORBA und auch zu der Vorentwicklung von MiRPA-X [115]).

Wenn der „machine monitoring“-Prozess den Token besitzt, prüft er, ob die festgelegte Anzahl an Hardwareinterrupts tatsächlich eingetroffen ist. Ist das der Fall, überträgt er ein Synchronisierungssignal (QNX-Puls) an die übergeordnete Steuerungsebene. Der Steuerungskern („control core“, s. Abschnitt 5.1.2) wird temporär auf hoher Prioritätsebene aktiv und realisiert daraufhin, wie beschrieben, im Wechsel einen von zwei Vorgängen:

**Ausführung der Bewegungs- und Sensormodule** Die Bewegungs- und Sensormodule werden benachrichtigt, dass ihre Algorithmen für den kommenden Steuerungszyklus zur weiteren Bearbeitung des aktiven Aktionsprimitives benötigt werden (s. Abschnitt 5.1.2).

**Auswertung der Bewegungs- und Sensormodule** Die Zustände der Bewegungs- und Sensormodule werden ermittelt. Es wird entschieden, ob die neuen Daten freigegeben werden können, ein Fehler aufgetreten ist oder ein Modul nicht erfolgreich beendet werden konnte (s. Abschnitt 5.1.2).

Mit diesem Verfahren, den übergeordneten Steuerungszyklus weiter zu unterteilen, ist eine Möglichkeit vorhanden, auch die nachrichtenbasierte Kommunikation auf der mittleren Steuerungsebene in einem zeitlichen Determinismus zu realisieren. Der „monitoring and coordination“-Prozess ist die Schnittstelle zur obersten Steuerungsebene. Er realisiert die Fehlerbehandlungsalgorithmen und empfängt die Aktionsprimitive vom „robot program interpreter“-Prozess.

In der oberen Steuerungsebene befinden sich auch der Serverprozess für die grafische Benutzerinteraktion, die Visualisierung von Steuerungsdaten und weitere Dienste zum Zugriff auf allgemeine Ressourcen des Rechners. Diese Steuerungsebene verwendet ereignisgesteuerte (nicht-zyklische) Schedulingprinzipien, um die bei der Bearbeitung der beiden erstgenannten Ebenen verbleibende Rechenzeit auf die einzelnen Prozesse zu verteilen.

### Software in the Loop

Da die beiden oberen Ebenen der Steuerung (s. Abbildung 5.9) roboterunabhängig sind, ist es möglich, das Verhalten der Steuerung auch mit dem „control core“-Prozess allein zu simulieren. Dazu werden die Prozesse „fieldbus communication“ und „subordinated drive control“ unabhängig von den oberen Steuerungsebenen betrieben. Auf diese Weise können die Algorithmen der Steuerung, wie z.B. das komplexe Aufstarten des Roboters, sehr einfach getestet werden. Das gleiche gilt auch zur Inbetriebnahme der beiden oberen Ebenen der Steuerung: Der „robot program interpreter“-Prozess kann betrieben werden ohne die Notwendigkeit, den Roboter auch tatsächlich zu bewegen oder überhaupt einen Roboter angeschlossen zu haben. Eine Simulationsumgebung ermöglicht die Visualisierung des Roboters und stellt gleichzeitig ein Umgebungsmodell für den Roboter zur Verfügung, das die erforderlichen Sensorwerte generiert. Auf diese Weise lässt sich der gesamte Steuerungskern in der Simulationsschleife testen. Das bedeutet also, dass der Steuerungsentwickler die Steuerung komplett mit der

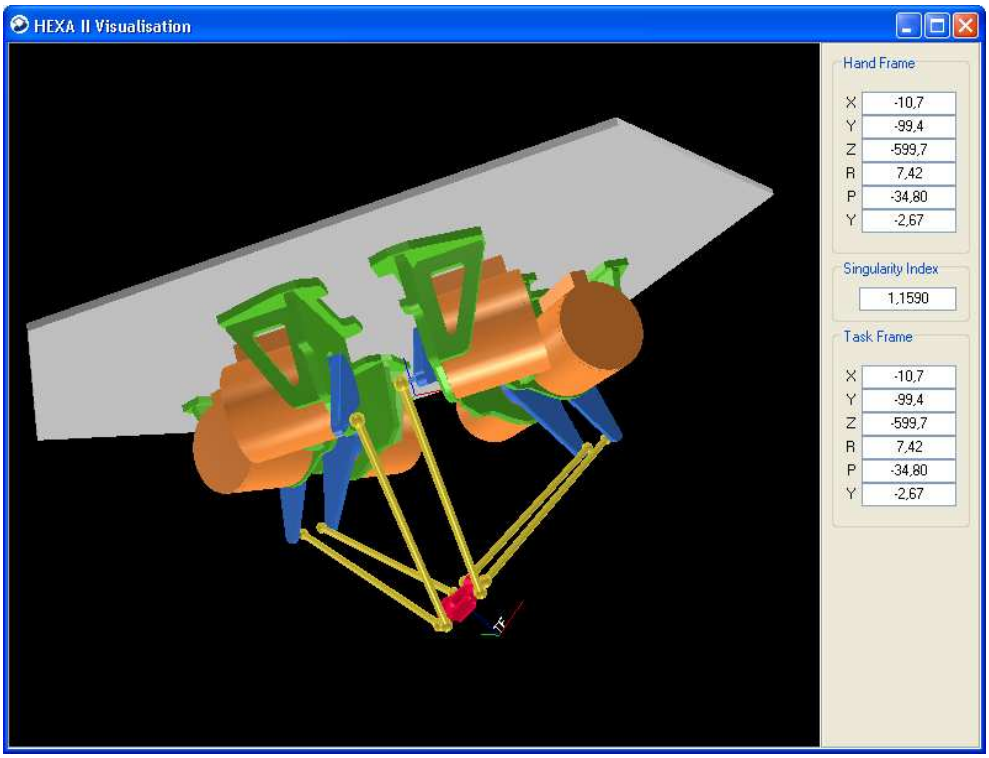



Abbildung 5.10: Simulationsumgebung für den HEXA

Originalsteuerung selbst simulieren kann. Die Simulationsumgebung für den HEXA-Roboter ist in Abbildung 5.10 dargestellt. Sie wird verwendet, um in Sequenzen von Roboterbewegungen nachzuprüfen, ob und wie nahe direkte kinematische Singularitäten angefahren werden; dazu dient ein Singularitätsindex, der in der Mitte der Statusanzeige dargestellt ist.

### 5.1.3 Informationstechnische Struktur

In Abschnitt 4.4 wurde die Hardware zur informationstechnischen Verbindung der Roboterkomponenten allgemein vorgestellt. In den folgenden Absätzen erfolgt die Darstellung der Komponenten, die im Zusammenhang mit den Versuchsträgern im Einzelnen entwickelt wurden, um darüber auf Sensor- und Aktordaten zugreifen zu können. Abbildung 5.11 zeigt die schematische Gesamtstruktur des HEXA Versuchsträgers aus kommunikationstechnischer Sicht.

#### Schnittstelle für Antrieb

Jeder der in Abbildung 5.11 dargestellten Antriebsverstärker (insgesamt sind es 6), ist für einen Antrieb des HEXA zuständig und wird über einen eigenen Kommunikations-

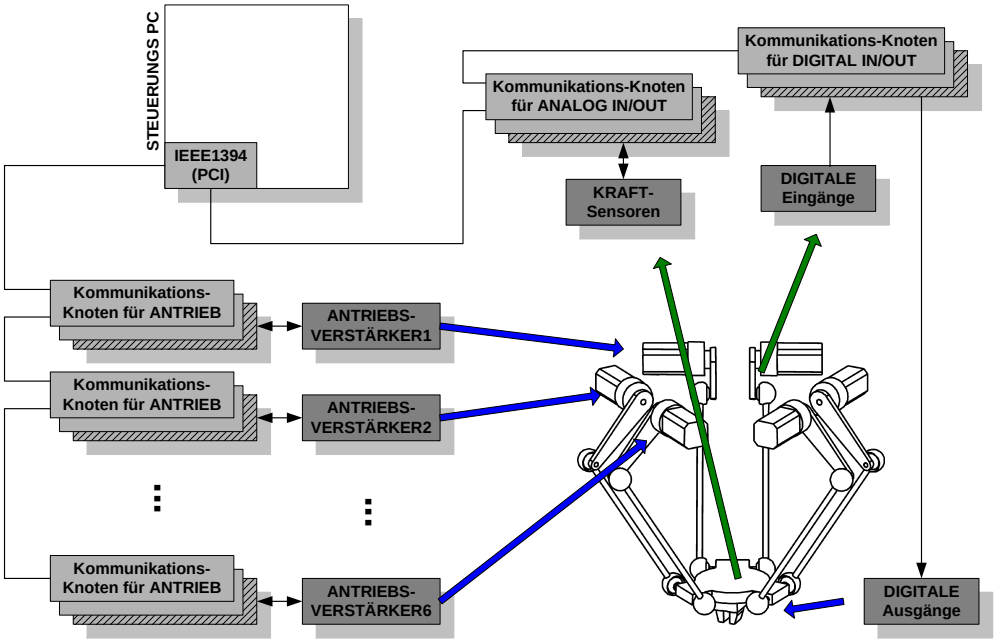



Abbildung 5.11: Schematischer Aufbau des HEXA

knoten informationstechnisch mit dem Steuerungssystem verbunden. Die Antriebsverstärker kommunizieren dabei über eine Dual-Port-RAM-Schnittstelle (DPRAM) mit dem Kommunikationsknoten, der mit einem DSP (TSB 320C6211) realisiert wurde und folgende weitere Baugruppen enthält:

**Modifiziertes IEEE1394-Modul** Zur Anpassung an den in den Antriebsverstärkern befindlichen Erweiterungsschacht wurde die Grundplatine aus Abbildung 4.48 unter Beibehaltung der technischen Eigenschaften baulich modifiziert. Lediglich die Anzahl von ursprünglich 3 Ports musste wegen mangelndem Bauraum auf 2 reduziert werden.

**Dual-Port-RAM-Modul** Der Antriebsverstärker Servostar SR610 der Firma Danaher Motion, der für die Ansteuerung der Anriebe des HEXA verwendet wird, nutzt zur Übertragung von Soll-, Ist- und Parameterdaten nach außen eine Dual-Port-RAM-Schnittstelle (DPRAM). Dafür wurde eine Schaltung mit passendem DPRAM-IC und entsprechender Hilfsbeschaltung entwickelt.

Abbildung 5.12 zeigt die Realisierung des Kommunikationsknotens, der einfach in den Erweiterungsschacht des Antriebsverstärkers eingeschoben werden kann. Da die im Antriebsverstärker vorhandene Versorgungsleistung nicht ausreicht, um den dargestellten Kommunikationsknoten zu versorgen, muss eine zusätzliche externe Versorgung von 5V appliziert werden.

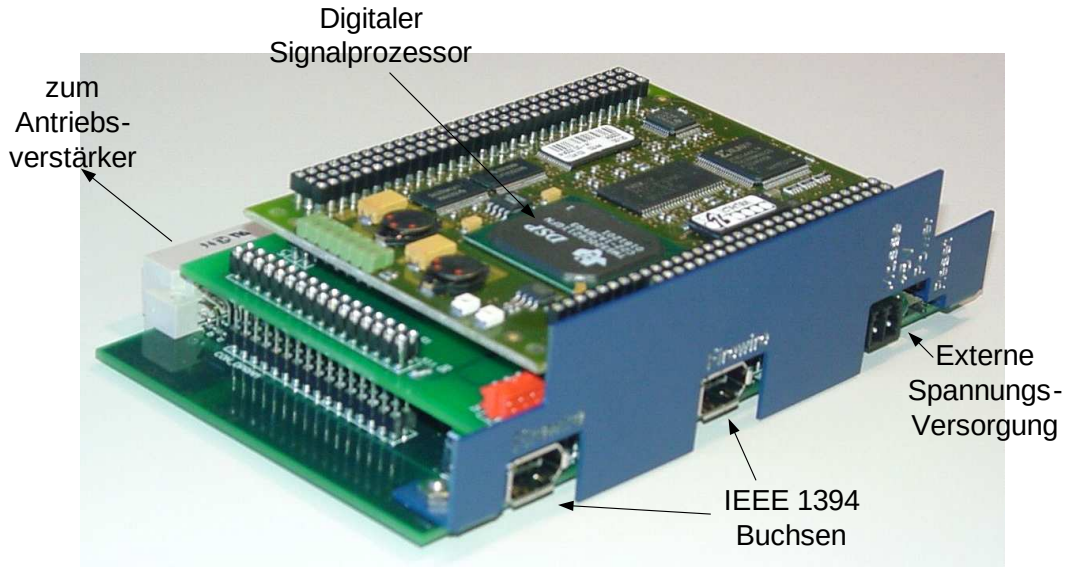



Abbildung 5.12: Schnittstellenhardware für den Antriebsverstärker

### Schnittstelle für Schaltschrankmontage

Um digitale Signale (z.B. am Roboter befindliche Endschalter) in der Steuerung abbilden zu können oder binäre Schalter zu realisieren, wurde auch ein digitales Ein-/Ausgabemodul (in Abbildung 5.11 rechts oben schematisch dargestellt) entwickelt und gefertigt. Es bietet jeweils 16 digitale Ein- und Ausgänge an, die über steckbare Verbinder mit der gewünschten Signalquelle oder -senke verbunden werden können und intern optisch entkoppelt sind. Dabei wird auch die gewünschte Ausgangsspannung über eine äußere Referenz vorgegeben, so kann dieses Modul auch übliche SPS-Signale verarbeiten. Zur Sicherheit ist im digitalen Ein-/Ausgabemodul auch eine Watchdogfunktionalität realisiert, die im Fehlerfall (z.B. beim Abbruch der Kommunikation oder Eintreten eines Programmfehlers) zum Öffnen eines Not-Aus-Kontaktes verwendet wird.

Abbildung 5.13 zeigt den Kommunikationsknoten für solch ein digitales Ein-/Ausgabemodul, von denen mehrere im System vorhanden sein können. Die mittlere Platine des dargestellten Stapels ist das DSP-Modul, das die notwendigen Softwaremodule enthält und ausführt. Die Aktualisierungsfrequenz für die digitalen Signale ist abhängig von der laufenden Applikation und dem Arbeitszyklus des Kommunikationsprotokolls.

### Schnittstelle für analoge Ein- und Ausgabe

Ein weiterer Kommunikationsknoten sorgt für die Analog-Digital-Wandlung beliebiger Signale und die daran anschließende Übertragung der digitalisierten Spannungswerte an die steuernde Applikation auf dem zentralen Rechner bzw. die Digital-Analog-Wandlung von dort empfangener Daten. Mit Hilfe des AD-Knotens werden

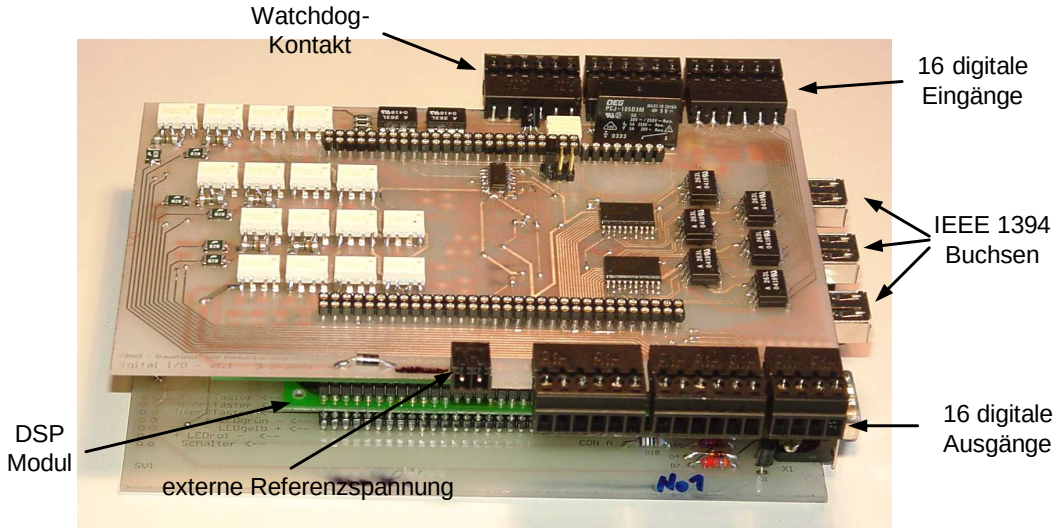



Abbildung 5.13: Schnittstellenhardware für digitale IO

die Kraftsensoren am Greifer und die adaptronischen Komponenten in der Roboterstruktur ausgewertet und damit der Bewegungszustand des Roboters genau erfasst. In der anderen Richtung werden über die DA-Wandlung dieses Knotens die als Aktoren nutzbaren adaptronischen Komponenten angesteuert.

Abbildung 5.14 zeigt einen solchen AD-Knoten, der hier aus drei stapelbaren AD-Modulen, der dazugehörigen Energieversorgung, Kommunikation und Recheneinheit besteht. Jede der dargestellten AD-Modul hat folgende technische Eigenschaften:

- 25-poliger Anschlussstecker für Sensoren
- 6 analoge Eingangskanäle, +/- 10V, 16 Bit Auflösung
- 2 analoge Ausgangskanäle, +/- 10V, 16 Bit Auflösung

Das AD-Modul wurde am Institut für Regelungstechnik der TU Braunschweig entworfen, entwickelt und gefertigt.

### Software für Schnittstellenmodule

Die in den oberen Absätzen vorgestellten Kommunikationsknoten werden über den IEEE1394-Bus miteinander und dem Steuerungs-PC verbunden. Damit nun die analogen und digitalen Sensordaten sowie die Istwerte der Antriebe den Softwaremodulen in der Steuerung bzw. die neuen Sollwerte und Parameter den Antrieben, digitalen Ausgängen und DAU-Knoten zugänglich gemacht werden können, sind weitere Kommunikationsfunktionalitäten auf Softwareebene implementiert. So besteht die Software auf jedem Schnittstellen-Knoten aus den folgenden interagierenden Modulen:

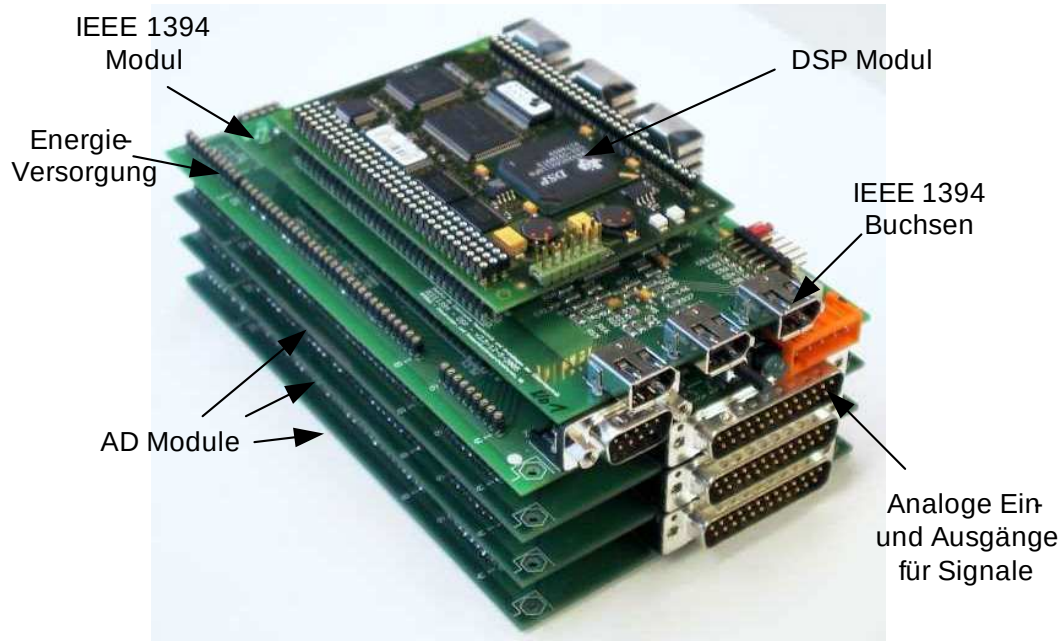



Abbildung 5.14: Schnittstellenhardware für AD-Knoten

**IEEE1394-Treiber** Dieser Treiber sorgt dafür, dass ankommende Datentelegramme vom Bus gelesen und in die Speicherbereiche der Anwendung geschrieben werden bzw. Daten von der Anwendung wieder auf den Bus gelangen.

**Applikation** Diese Software ist applikationsspezifisch und legt die Interpretation ankommender und ausgehender Daten fest. Im Fall des Antriebsverstärker-Interface wird hierin ein spezielles Protokoll zur Kommunikation mit dem Antriebsverstärker festgelegt oder aber, abhängig von empfangenen Daten, der Antriebsverstärker konfiguriert bzw. mit neuen Sollwerten für die Motorpose beaufschlagt. Für andere Knoten ist die Funktionalität entsprechend.

**IAP** Das IAP ist das verbindende Softwaremodul. Es legt Identifizierungen für externe Komponenten fest und sorgt dafür, dass Daten zwischen Steuerung und Komponenten im geforderten zeitlichen Bezug übertragen werden und ihr Ziel garantiert erreichen.

Die Softwaremodule der Steuerung finden aktuelle Sensordaten auf diese Weise innerhalb der über MiRPA-X angemeldeten Shared-Memory-Bereiche wieder bzw. schreiben ihre Sollwerte in selbige. Das IAP erledigt die Verteilung der Daten innerhalb des Robotersystems.

### Übersicht über Datenflüsse

Ein Applikationsprofil dient zur ganzheitlichen, systematischen Beschreibung eines Systems aus kommunikationstechnischer Sicht. Es berücksichtigt dabei sowohl spezifische Applikationsfunktionalitäten des zu beschreibenden Systems als auch die

zur Realisierung dieser Funktionalitäten notwendigen Kommunikationsmechanismen des verwendeten Kommunikationssystems. Mit dem Applikationsprofil kann auf diese Weise die Veranschaulichung einer Systemübersicht, die Ermittlung der zeitlichen und kommunikationstechnischen Erfordernisse und auch die Lokalisierung kritischer Applikationsbereiche dargestellt werden. Die Veranschaulichung geschieht dabei unter Nutzung oder in Anlehnung an den UML Standard.

Abbildung 5.15 zeigt das für die externen Komponenten des Demonstrators TRIGLIDE [133] entwickelte Applikationsprofil in Form eines Assembly-Diagramms der SysML[2]. Dabei wird jedes realisierte Hardwaremodul (als einzelne elektronische Schaltung) gemeinsam mit seinen Schnittstellen (kleine Quadrate) zu anderen Modulen dargestellt. Der verwendete Kommunikationsmechanismus basiert auf der Nutzung von Shared-Memory; es erfolgt also ein bedarfsorientiertes Kopieren der Daten zwischen Speicherbereichen unterschiedlicher Softwareebenen bzw. Geräten.

Um die Datenflüsse und die dabei verwendeten Kommunikationsmechanismen innerhalb der Steuerung auf dem Steuerungsrechner zu verdeutlichen, ist in Abbildung 5.16 ein weiteres Assembly-Diagramm dargestellt. Dort werden, die Synchronisationsmechanismen für den Shared-Memory-Zugriff ausgenommen, alle in der Steuerung verwendeten Kommunikationsmechanismen von MiRPA-X mit unterschiedlichen Linientypen dargestellt. Weiter wird die Anwendung in unterschiedliche Steuerungsebenen eingeteilt, um die Unterscheidung von unterlagerter Echtzeitregelung, modulbasierter Regelung und Steuerung zu ermöglichen. Die schwarz umrandeten Softwaremodule sowie die schwarzen Linien haben immer einen direkten Bezug zu den Echtzeitaktivitäten jedes Regelzyklus. Grau gezeichnete Umrisse und Linien verdeutlichen den Bezug zu Steuerungsfunktionalitäten der Soft-Realtime-Ebene. Bei nachrichtenbasierten Kommunikationsmechanismen enthält ein Kommunikationsendpunkt (Port) den Buchstaben „S“ bzw. „C“ um zu verdeutlichen, dass es sich lokal um eine Server- oder Clientfunktionalität handelt. In Abbildung 5.16 ist erkennbar, dass zur Realisierung der Echtzeitfähigkeit innerhalb der unterlagerten Antriebsregelung ausschließlich deterministische Kommunikationsmechanismen verwendet werden, die unabhängig vom aktiven Eingriff des ObjectServers arbeiten.

Mit Hilfe des Applikationsprofils war es möglich, den Umfang erwarteter Kommunikationsvorgänge auf dem Steuerungsrechner im Voraus zu bestimmen und eine Kommunikationsablaufplanung in der Weise zu realisieren, dass im Betrieb auftretende unlösbare kommunikationstechnische Abhängigkeiten (Deadlocks) innerhalb der Datenflüsse im Vorfeld bei der Realisierung der Softwaremodule und deren Konfigurierung vermieden werden konnten. Innerhalb der zentralen Steuerung kommen im zyklischen Betrieb ausschließlich Shared-Memory-Mechanismen und datenlose Nachrichten zur Synchronisierung zum Einsatz, so dass eine Bandbreitenangabe hier entfällt.

### 5.1.4 Experimentelle Ergebnisse

In diesem Abschnitt werden die experimentellen Ergebnisse vorgestellt, die mit der oben beschriebenen Steuerungsarchitektur und den implementierten Regelungsalgo-

[2] SYStem Markup Language

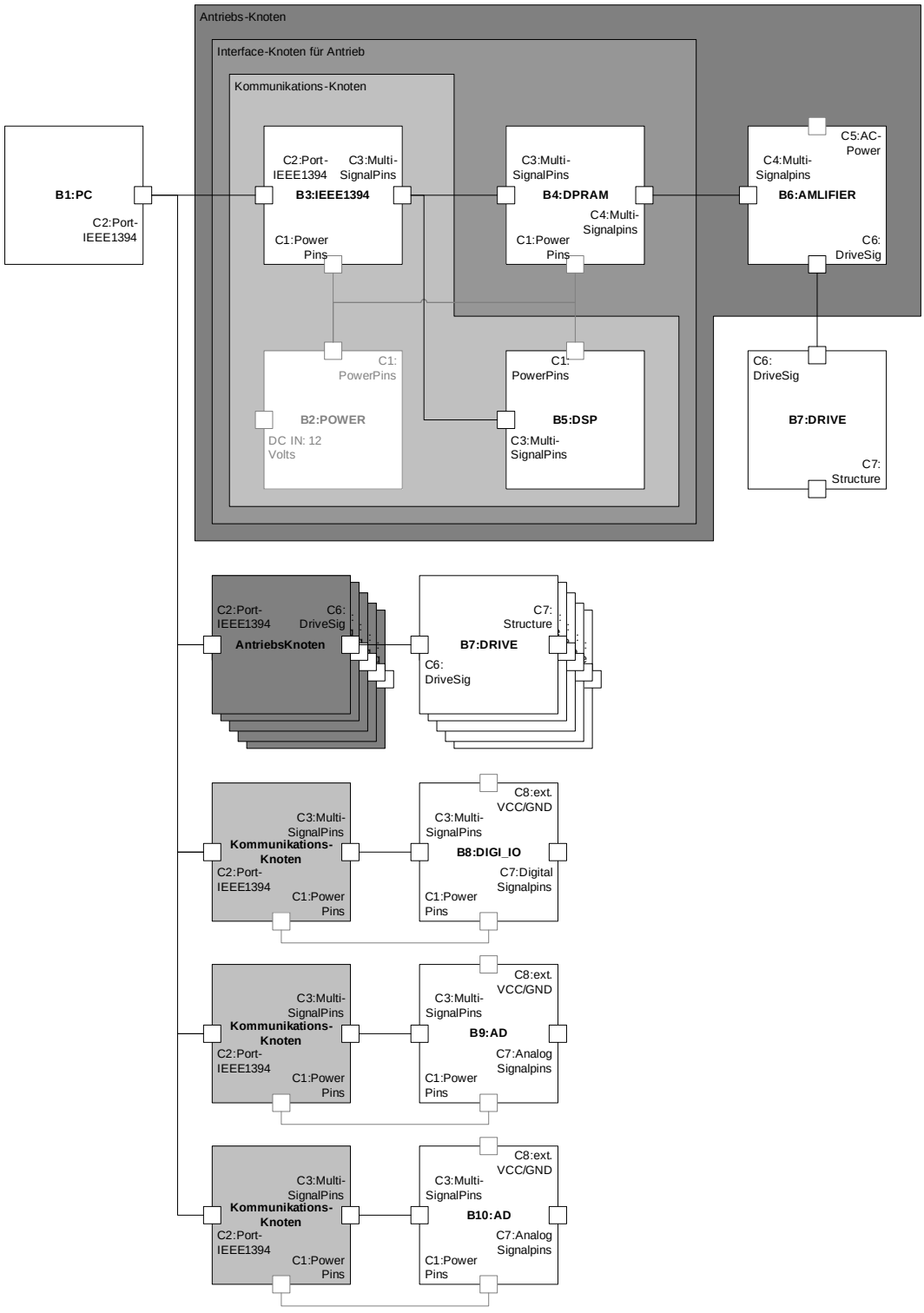



Abbildung 5.15: Applikationsprofilteil für externe Hardwaremodule

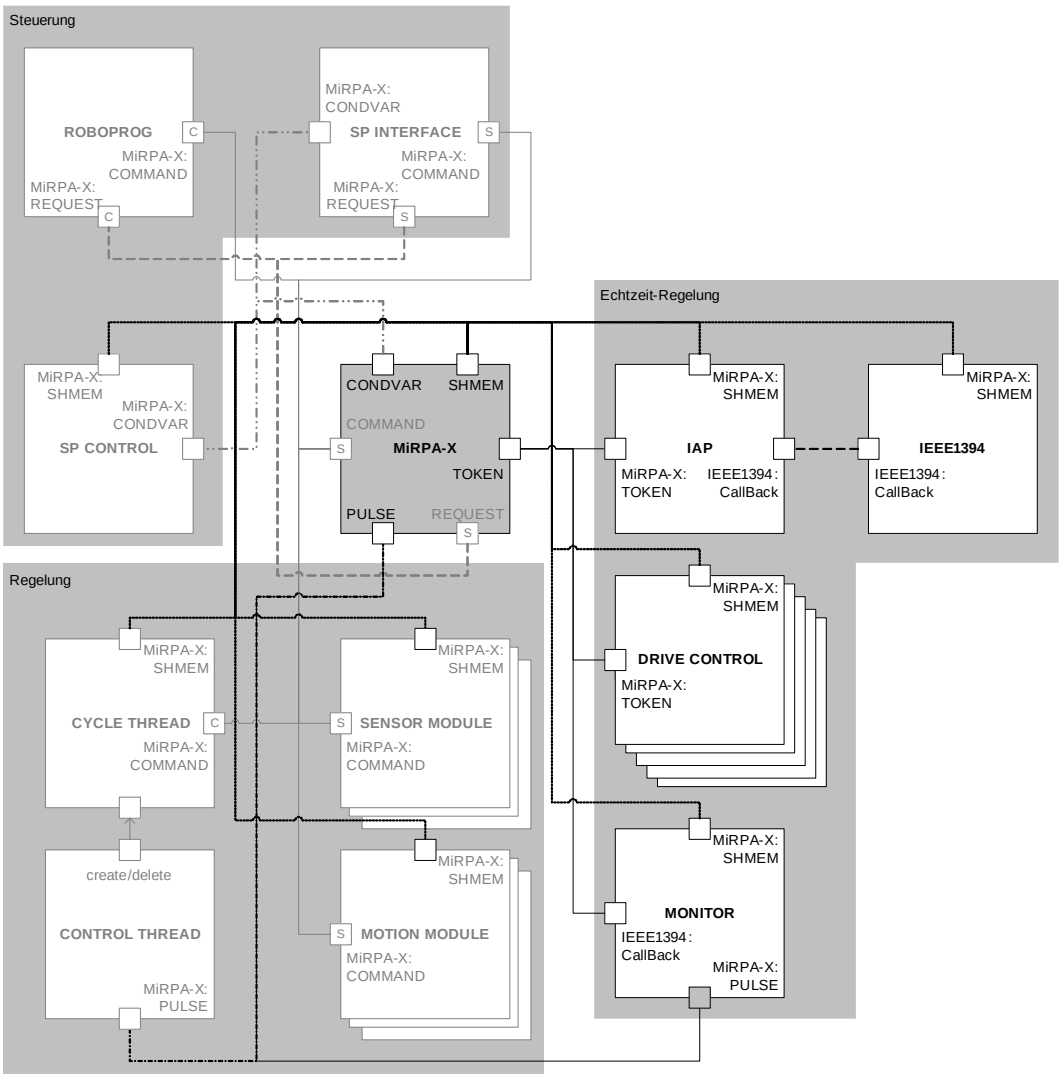



Abbildung 5.16: Applikationsteil für die zentralen Softwaremodule

rithmen auf dem HEXA-Roboter erzielt wurden. Die Zielplattform war dabei ein handelsüblicher PC mit AMD Athlon 64 3000+ CPU und 1 GB RAM. Das verwendete Betriebssystem war QNX Neutrino in der Version 6.2.1.

Da die im HEXA-Roboter verwendeten Antriebsverstärker nicht in der Lage waren, schneller als in einem $4kHz$-Takt (alle $250\mu s$) Daten auszutauschen, wurde dieser Kommunikationszyklus als Grenze für die untere Steuerungsebene festgelegt. Mit dieser Zyklusfrequenz ist die geschlossene Regelung über den Bus maximal sinnvoll und möglich. Der Bahnplaner arbeitet dann in einem Zyklustakt von $1kHz$ ($1ms$). Diese erwähnten Zyklusfrequenzen wären ausreichend, um mit dem Endeffektor des HEXA-Roboters Beschleunigungen von $10g$ und Geschwindigkeiten von $5m/s$ zu erreichen, ohne dabei einen Folgefehler von mehr als $1,3°$ in den Antrieben hervorzurufen.

Abbildung 5.17 zeigt einen typischen Steuerungszyklus, bei dem ein Positionsplaner als Bewegungsmodul eine ruckbegrenzte Trajektorie generiert, die der Steuerungskern auf dem Antrieb regelt. Dabei bearbeitet der Steuerungskern einkommende Daten von einem Kraft/Momentensensor im Endeffektor. Alle Zeitmessungen sind mit dem „Instrumented Kernel“ von QNX aufgezeichnet und dem dazugehörigen Analysewerkzeug ausgewertet.

In der untersten Schedulingebene der Steuerungsarchitektur wird jeder neue Start eines Kommunikations- bzw. Regelungszyklus durch einen Puls von der IEEE1394-Hardware signalisiert (blaue, gestrichelte Linie). Zunächst kommt dann der IEEE1394-Treiber zusammen mit dem Kommunikationsprotokoll IAP zum Ablauf, um empfangene Sensordaten in den Prozessspeicher des Rechners abzulegen. Innerhalb des Steuerungskerns wird anschließend eine Arbeitsraumüberwachung (zum Prüfen auf Singularitäten) und ein vorsteuernder Antriebsregler mit realisiertem „Computed Torque“-Algorithmus zur Ausführung gebracht. Der Ablauf dieser drei Prozesse wird über den Tokenmechanismus des Schedulers von MiRPA-X organisiert (s. (a) in der Abbildung). Ein einzelner Applikations-Prozesswechsel beansprucht dabei auf dem erwähnten Entwicklungsrechner rund $4\mu s$ mit einem maximalen Jitter von $0,8\mu s$.

Wie in Abschnitt 5.1.2 dargestellt, aktiviert der Monitorprozess anschließend den „control thread“ über einen Puls (s. (b) in der Abbildung). Beide Prozesse, der „control thread“ und der „cycle thread“, befinden sich in der Koordinationseinheit des Steuerungskerns, also in der mittleren Steuerungsebene. Der „control thread“ prüft zunächst, ob ein neues Aktionsprimitiv empfangen wurde oder ob der Nutzer das Roboterprogramm beenden möchte. Danach wird in den entsprechenden Shared-Memory-Bereichen markiert, von welchen Sensoren Informationen für den aktuellen Steuerungszyklus benötigt/erwartet werden (grüner Pfeil in der Abbildung). Ein Broadcast-`COMMAND` wird abgesetzt, welcher von der Middleware an alle vorhandenen Sensormodule (hier nur eines dargestellt) weitergeleitet wird. Sobald ein Sensormodul aktiv wird, prüft es anhand des SharedMemory-Bereiches, ob es seinen Algorithmus anwenden muss und quittiert den Erfolg wieder im SharedMemory. Die Gesamtheit dieser Vorgänge benötigt $35\mu s$. Davon benötigt das Sensormodul selbst $13\mu s$ und führt darin eine mittelwertbildende Rauschunterdrückung und die Transformation der Kräfte und Drehmomente in den TaskFrame durch. Wenn alle Sensoren auf diese Weise abgefragt wurden, sendet der „cycle thread“ einen `COMMAND` zum Positionsplaner-Modul. Der Zeitaufwand für die Kommunikation, um ein solches Bewegungsmodul

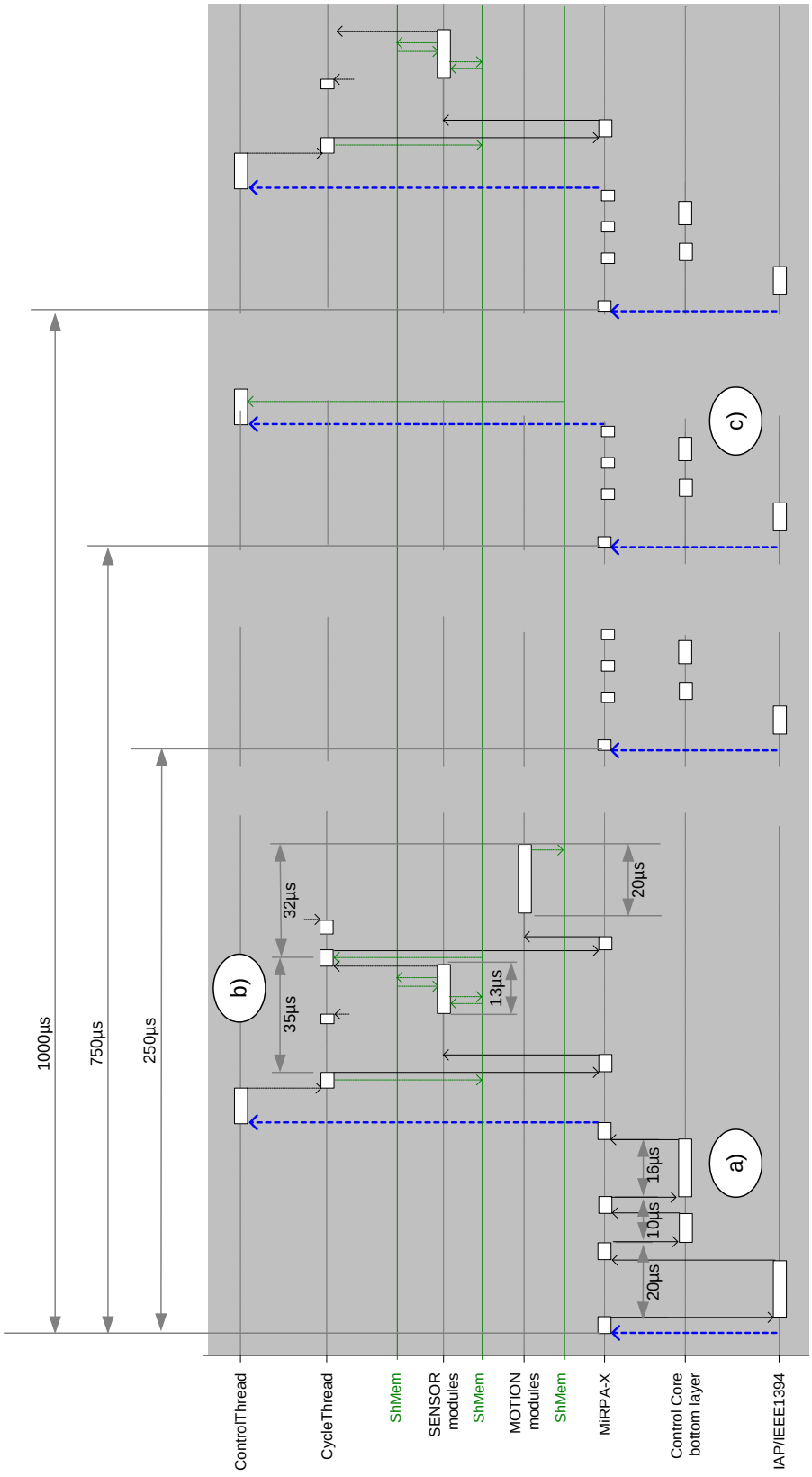



Abbildung 5.17: Messergebnisse für den HEXA-Roboter

| Funktion (1000 samples) | Mittelwert ($\mu s$) | Maximalwert ($\mu s$) | Standard-abweichung |
|---|---|---|---|
| SendCommand (2kByte) | 7,49 | 8,08 | 0,26 |
| SendRequest (2kByte) | 8,91 | 9,89 | 0,29 |
| CheckCondition | 0,13 | 0,29 | 0,04 |
| SignalCondition | 1,33 | 1,46 | 0,04 |

Tabelle 5.1: Gemessene Ausführungszeiten von MiRPA-X-Funktionen

vollständig abzufragen, beträgt $12\mu s$. Alle Kommunikationsvorgänge und Applikationsalgorithmen sind für dieses Beispiel innerhalb von $130\mu s$ abgeschlossen, so dass noch große zeitliche Reserven zur Verkleinerung des Steuerungszyklus oder für komplexere Algorithmen zur Verfügung stünden.

Nach $750\mu s$, d.h. nach dem dritten von vier Regelungszyklen, wird der „control thread“ wieder aktiviert und prüft, ob alle Bewegungsmodule erfolgreich abgelaufen sind, führt dann die notwendigen Berechnungen für den Task-Frame-Formalismus durch und prüft weiter, ob die Stopp-Bedingung für das Aktionsprimitiv eingetreten ist. Danach wird die neue Trajektorieninformation in den Shared-Memory-Bereich für den unterlagerten Antriebsregler eingetragen (s. (c) in der Abbildung). Im nächsten Regelungszyklus wird diese Information dann von dem unterlagerten Antriebsregler berücksichtigt, so dass sich der Roboter entsprechend bewegt.

In Tabelle 5.1 sind die wichtigsten Ausführungszeiten für die verwendeten Middlewarefunktionalitäten dargestellt. Diese Zeiten wurden auf dem gleichen Rechner gemessen, auf dem auch die oben beschriebene Steuerung abgelaufen ist, und zwar jeweils mit 1000 Wiederholungen. Auffallend sind die nur kleinen Differenzen zwischen den mittleren und den maximalen Ausführungszeiten und der sehr kleine Jitter, ausgedrückt durch die Standardabweichung.

# 6 Ausblick

### Verteilte Steuerung

Zukünftig soll die in dieser Arbeit vorgestellte Steuerung für Parallelrobotersysteme auf mehrere Rechner verteilt werden, um vorrangig komplexere und rechenzeitintensivere Bewegungsalgorithmen realisieren zu können. Auch die Integration von Kamerasystemen in diese Steuerung könnte dadurch ermöglicht werden, um Montage- oder Fügevorgänge optisch zu steuern. Die Strategie ist dabei, eine Kommunikations-Infrastruktur auf Basis der verteilten Middleware **MiRPA-XD**[1] zu entwickeln, so dass die Ausführung von Funktionseinheiten auf unterschiedlichen Rechnern transparent wird. Der auf dem IEEE1394-Standard basierende und über das IAP für Roboteranwendungen nutzbar gemachte Feldbus bietet dabei sehr gute Voraussetzungen für beides, Kommunikationsbandbreite und Synchronität mehrerer Rechner. Es ist aber weiter zu untersuchen, ob gerade für die Rechner-Rechner-Kommunikation andere Kommunikationssysteme möglicherweise besser geeignet sind als FireWire, so dass letztlich auch mehrere Kommunikationskonzepte gleichzeitig zur Anwendung kommen könnten.

Erste Schedulability-Analysen der Steuerungsalgorithmen für den Parallelroboter HEXA zeigen, dass der größte Nutzen der angestrebten Verteilung erreicht werden kann, wenn die Algorithmen der Bewegungsmodule sowie des sehr rechenintensiven Maschinenmonitors auf separaten Rechnern ablaufen würden. Die verteilten Maschinenmonitor-Algorithmen würden dann in einem Token-Passing-Zyklus eines jeweiligen Rechners ablaufen und von jedem neuen Kommunikationszyklus auf dem gemeinsamen Kommunikationsbus synchronisiert werden.

Um die Bewegungsmodule auf mehrere Rechner zu verteilen, bietet sich für die Umsetzung innerhalb der verteilten Middleware das Stub/Skeletonprinzip (z.B. aus CORBA bekannt) an. Unter Beibehaltung der dargestellten Fehlerimmunität könnte ein „Skeleton“ des Bewegungsmoduls innerhalb der mittleren Steuerungsebene (s. Abbildung 5.9) ausgeführt werden: Wenn ein Skeletonmodul-Algorithmus vom Steuerungskern angefragt wird, könnte es sich autonom mit seinem „Stub“-Prozess auf einem entfernten Rechner verbinden und so lange blockiert bleiben, bis die Antwortnachricht von dem Stub eintrifft. Dieser Ansatz ermöglicht auch den Einsatz fortgeschrittener Konzepte wie z.B. den Ausgleich von Auslastungsspitzen einzelner Recheneinheiten und Optimierungen am Steuerungssystem zur Laufzeit.

Mit dem Verteilungskonzept ergibt sich die Frage, wann eine Verteilung einzelner Bewegungsmodule in einem aktuellen Steuerungskontext überhaupt sinnvoll und prak-

[1] MiRPA-X distributed

tikabel ist und wie eine mögliche Online-Auslagerung dann praktisch erfolgen könnte. Sobald der zeitliche Aufwand für die lokale (sequentielle) Organisation und Ausführung der Bewegungsmodule die verfügbare Zykluszeit übersteigt, könnte eine Auslagerung ausgewählter Bewegungsmodule sinnvoll sein, sofern der lokale (zusätzliche) zeitliche Aufwand für Verwaltungs- und Kommunikationsaufgaben die zeitlichen Anforderungen insgesamt nicht negativ beeinflusst. Die Prüfung für eine jeweilige Steuerungs-Modulkonfiguration muss im Einzelnen erfolgen und gestaltet sich besonders komplex, wenn die zeitlichen Anforderungen der Bewegungsmodule zur Laufzeit aufgabenabhängig variieren. Die Entwicklung erfolgreicher Strategien zur Verteilung von Bewegungsmodulen ist daher notwendig und muss unter Berücksichtigung formaler Analysen erfolgen.

### Exemplarische Kommunikationsstruktur

In Abbildung 6.1 ist ein exemplarischer Steuerungszyklus abgebildet. Das Beispiel lehnt sich an die im SFB 562 gewählte zentrale Steuerungsarchitektur (s. Abschnitt 5.1.2) an und zeigt, wie MiRPA-XD genutzt werden kann, um die verwendeten Bewegungsmodule der zentralen Steuerung auf einen anderen Rechner (Modulrechner) zu verteilen. Dabei wird auch der Einfluss auf andere Softwaremodule (vor allem IAP) dargestellt.

In der Abbildung sind die Modulabläufe auf dem Steuerungsrechner (links) und dem parallel arbeitenden Modulrechner (rechts) abgebildet. Dabei wird jedem Modul eine eigene Lebenslinie (durchgezogen) zugeordnet. Um den Datentransfer der einzelnen Softwaremodule zu verdeutlichen und dabei gleichzeitig die spezifische Speicherfunktion für Applikationsdaten oder MiRPA-XD-Parameterdaten unterscheiden zu können, wurden jeweils zentral bei MiRPA-XD drei gestrichelte Linien angeordnet, die den nicht freigegebenen Shared-Memory (1), den allgemeinen Shared-Memory (2) und den Status- bzw. die Konfigurationsspeicher von MiRPA-XD (3) symbolisieren. Die grau eingefärbten Aktivitätsblöcke stellen Module dar, die zur Realisierung der verteilten Middlewarefunktionalität verändert oder neu entwickelt werden müssen. Die gestrichelt umrandeten Blöcke sind abhängig von den Bewegungsmodulen und ihren Ein- und Ausgabedaten.

Um trotz des Umfangs der Aktivitäten eine funktionsbezogene Übersicht zu erhalten, wurde auf eine äquidistante Darstellung verzichtet. Der zeitliche Abstand zwischen jeweils zwei aufeinanderfolgenden CSTs im IEEE1394-Modul ist also in der Realität stets konstant. Gedanklich muss die Darstellung also an entsprechenden Stellen gespreizt werden. Die Anzahl der Regelungszyklen ist korrekt; ist der Ablauf am unteren Ende des Diagramms angekommen, so wiederholt sich der Ablauf oben erneut. Im Folgenden wird der dargestellte Kommunikations- und Aktivitätsablauf entlang der ausgezeichneten Regionen (A bis K) beschrieben:

**A** Nachdem das IAP die über den IEEE1394 empfangenen Telegramme ausgewertet und in die entsprechenden Shared-Memory-Bereiche geschrieben hat, lesen die Module der unterlagerten Antriebsregelung die neuen Istwerte aus, berechnen die Sollwerte für den nächsten Regelungszyklus und schreiben sie wieder in

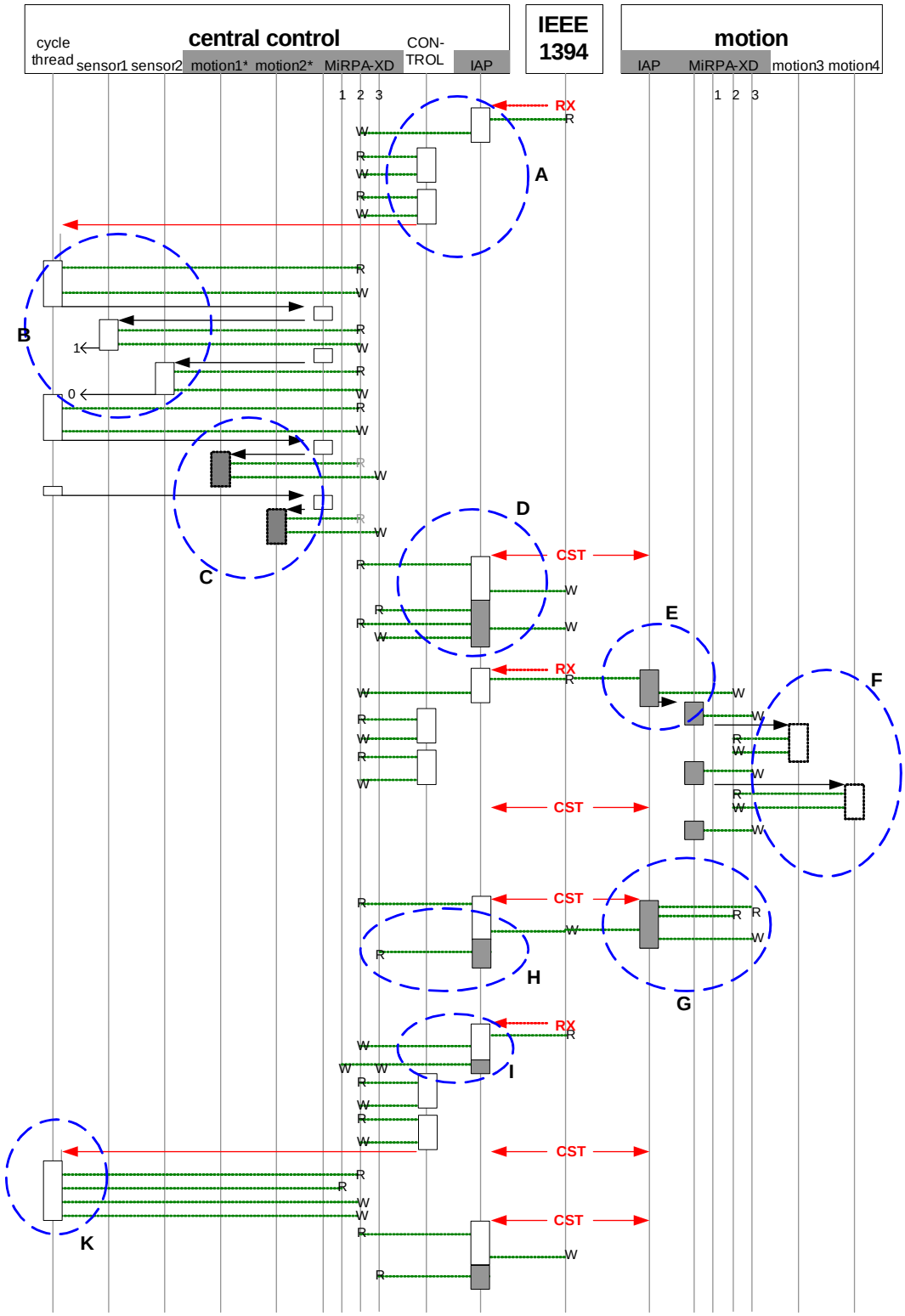



Abbildung 6.1: Exemplarischer Steuerungszyklus unter Nutzung verteilter Bewegungsmodule

den entsprechenden Shared-Memory. Gemäß des Steuerungsarchitekturkonzepts wird nun ein Aktivierungspuls für den Steuerungskern erzeugt.

**B** Der Steuerungskern (cycle thread) prüft zunächst, ob der vorhergehende Steuerungszyklus erfolgreich beendet wurde, setzt anhand des aktiven Aktionsprimitives fest, welche Sensoren zur Berechnung des nächsten Steuerungszyklus erforderlich sind und versendet anschließend ein Kommando, welches die entsprechenden Sensormodule zur Ausführung bringt. Die erfolgreiche Ausführung der Sensormodule wird anschließend überprüft.

**C** Wenn alle erforderlichen Sensormodule erfolgreich beendet wurden, entscheidet der Steuerungskern, welche Bewegungsmodule zur Trajektorienberechnung erforderlich sind und setzt die entsprechenden Konfigurationsvariablen. Über jeweils ein Kommando werden nun anstatt der Bewegungsmodule nur die Bewegungsmodul-Stubs (s. Java RMI) gestartet, welche die im Bewegungsmodul behandelten *Nachrichtennamen* und *Shared-Memory-Namen* für Ein- und Ausgabe in eine spezielle XD-Konfigurationsvariable speichern.

**D** Im Anschluss an das Versenden des MDT prüft das IAP anhand der XD-Konfigurationsvariablen, für welche Bewegungsmodule Aufträge hinterlassen wurden. Die entsprechenden Shared-Memory-Daten werden ausgelesen, als asynchrones Telegramm verschickt und das Verschicken anschließend in der XD-Konfigurationsvariablen quittiert.

**E** Das IAP auf dem Modulrechner empfängt das vom Steuerungsrechner versendete XD-Telegramm, schreibt die Daten in die entsprechenden Shared-Memory-Bereiche und bewirkt anschließend über das Versenden einer Kontrollnachricht an den ObjectServer (enthält eine Liste der aufzurufenden Kommandos), dass die lokalen Original-Bewegungsmodule aufgerufen werden.

**F** Die Bewegungsmodule lesen nun aus den lokalen Shared-Memory-Bereichen ihre erforderlichen Daten heraus, führen Berechnungen durch und schreiben entsprechende Ergebnisse wieder in die Shared-Memory-Bereiche zurück. Der erfolgreiche Ablauf aller Bewegungsmodule wird jeweils quittiert.

**G** Nun prüft das IAP den erfolgreichen Ablauf der Bewegungsmodule, liest die Ergebniswerte jeweils aus den Shared-Memory-Bereichen aus, schreibt sie in einem asynchronen Telegramm auf den Bus und bestätigt das erfolgreiche Versenden in einer XD-Konfigurationsvariable.

**H** Einschub: Auf dem Steuerungsrechner wird bei jedem MDT-Versand geprüft, ob es XD-Daten zu versenden gibt. Ist dies nicht der Fall, so werden keine XD-Daten versendet.

**I** Nachdem das IAP die DDT-Daten vom Bus gelesen und in die entsprechenden Shared-Memory-Bereiche geschrieben hat, wird auch das empfangene asynchrone XD-Telegramm ausgewertet und die Ergebnisse der Berechnungen vom Modulrechner werden in die entsprechenden Shared-Memory-Bereiche geschrieben.

**K** Nachdem die Funktionen der unterlagerten Regelung die neuen Sollwerte berechnet haben (s. A), prüft der Steuerungskern, ob die von den Bewegungsmodulen

berechneten Werte als für den folgenden Steuerungszyklus gültig angenommen werden sollen.

In dem dargestellten Anwendungsbeispiel zur Verteilung von Bewegungsmodulen sind die Sensormodule nur auf dem zentralen Steuerungsrechner vorhanden. Es ist natürlich auch denkbar, die Sensormodule auf allen beteiligten parallelen Modulrechnern ablaufen zu lassen. Da auf allen Rechnern die Istwerte der verteilten Komponenten gleichermaßen vorhanden sind, greifen damit alle Sensormodule auf gleiche Daten zu und liefern damit auch die gleichen Ergebnisse, die dann von den lokalen Bewegungsmodulen gelesen werden. Damit verringert sich das Datenaufkommen bei etwas gestiegenem Verarbeitungsaufwand. Diese Variationsmöglichkeit kann bei der Auslegung eines Steuerungssystems parametriert werden, um so das System gemäß unterschiedlicher Anforderungen zu optimieren.

### IAP und IEEE1394

Im SFB 562 wird weiterhin untersucht werden, inwiefern das IAP, wie es im Rahmen dieser Arbeit realisiert wurde, unabhängig von dem unterlagerten Kommunikationssystem eingesetzt werden kann. Zur Zeit ist ein Einsatz des IAP nur aufbauend auf dem IEEE1394-Standard möglich, allerdings zeichnen sich in der industriellen Kommunikation Entwicklungen ab, die den Einsatz der verschiedenen Industrial-Ethernet-Ansätze (Real-Time Ethernet) favorisieren (s. dazu auch Abschnitt 3). Um diesen Entwicklungen zu begegnen und das IAP universell einsetzbar zu machen, soll untersucht werden, auf welche Weise eine Hardware-Unabhängigkeit für das IAP erreicht werden kann.

In den bisherigen Arbeiten zur Bereitstellung von echtzeitfähiger Kommunikation innerhalb der Steuerung lag der Schwerpunkt auf der Konzipierung und Realisierung leistungsfähiger Softwaremodule unter Verwendung und Zusammenstellung von Standard-Hardwarekomponenten. Die entwickelte Infrastruktur ist mit ihrer Leistungsfähigkeit nun an Grenzen gestoßen, die nicht mehr über algorithmische Optimierungen, sondern ausschließlich durch Integration von zeitintensiven Softwareanteilen in schnelle Hardware überwunden werden können. Dazu müssen insbesondere die unteren Schichten des Kommunikationssystems auf diese Weise optimiert werden, um Latenzzeiten zu minimieren und damit geringere Zykluszeiten zu ermöglichen.

Am Ende des Abschnitts 4.4.2 wurde gezeigt, dass die Kommunikation in der derzeitigen Implementierung bei $2kHz$ Regeltakt immer noch ca. 40% der gesamten für die Steuerung verfügbaren Rechenzeit beansprucht. Dies liegt im Wesentlichen daran, dass die notwendigen zyklischen Kopiervorgänge während der Datenverarbeitung innerhalb der externen Kommunikationsknoten (IAP-Knoten) aufgrund der Nutzung des Mikrocontroller-Interfaces langsam sind. Der zeitliche Aufwand für den Empfang und den Versand von nur $20Bytes$ Nutzdaten auf einem IAP-Knoten beträgt derzeit ca. $40\mu s$. Durch die Integration von Vorverarbeitungsfunktionen der IAP-Knoten in einen Chip wird die Datenverarbeitungszeit innerhalb der IAP-Knoten erheblich reduziert. Dies führt unmittelbar zur Realisierungsmöglichkeit von Steuerungen mit

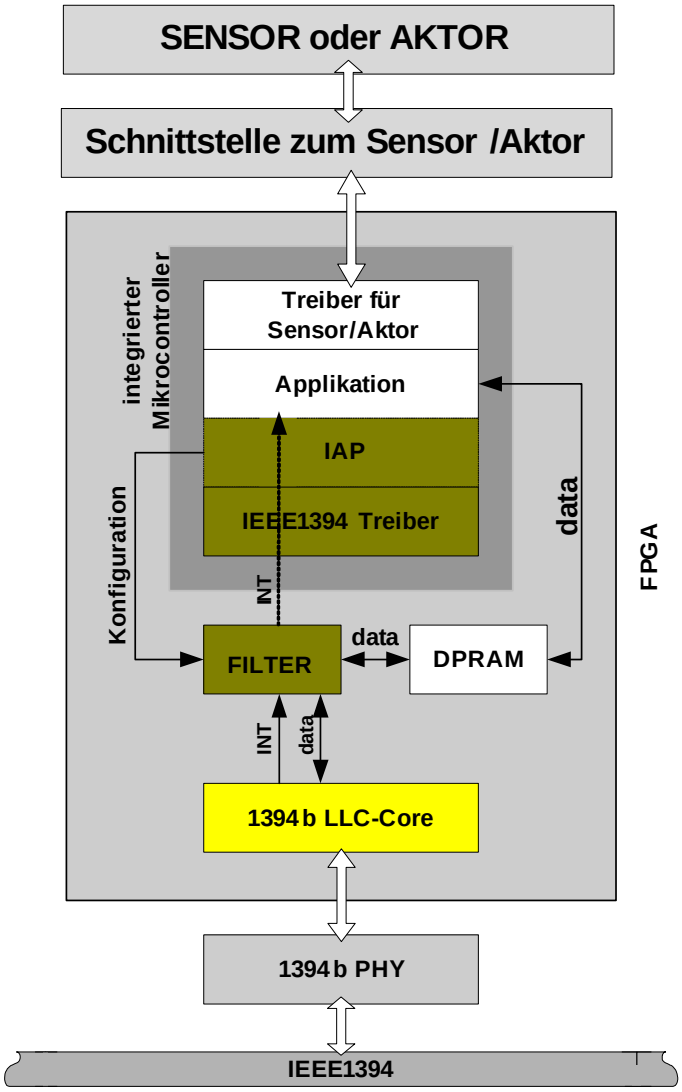



Abbildung 6.2: Integrierte Hardware-Struktur eines IAP-Knotens

einem Kommunikations-Rechenaufwand von deutlich unter 20%, selbst bei einem Regelungszyklus von $8kHz$.

Eine zusätzliche Optimierung der IEEE-1394-Hardware soll in naher Zukunft über den Einsatz des IEEE-1394b-Standards erfolgen. Die technischen Eigenschaften dieser Version des Standards sind in Abschnitt 3.3.1 ab Seite 73 dargestellt.

Abbildung 6.2 zeigt den Aufbau der geplanten integrierten Hardware eines IAP-Knotens. Als Integrations-Plattform wird ein FPGA (z.B. Xilinx Spartan 2E) gewählt. Über einen diskreten 1394b-PHY und dem integrierten 1394b-LLC-Core gelangen die empfangenen Daten über einen Filter-Automaten in das DPRAM. In einem integrierten Mikrocontroller kommen folgende Softwaremodule zur Ausführung: Das IAP organisiert die Kommunikation innerhalb des Knotens und greift über den IEEE-1394-Treiber auf die untergeordnete Kommunikationsschicht zu. Über den Sensortreiber schreibt bzw. liest die Applikation Soll- bzw. Istwerte der angeschlossenen

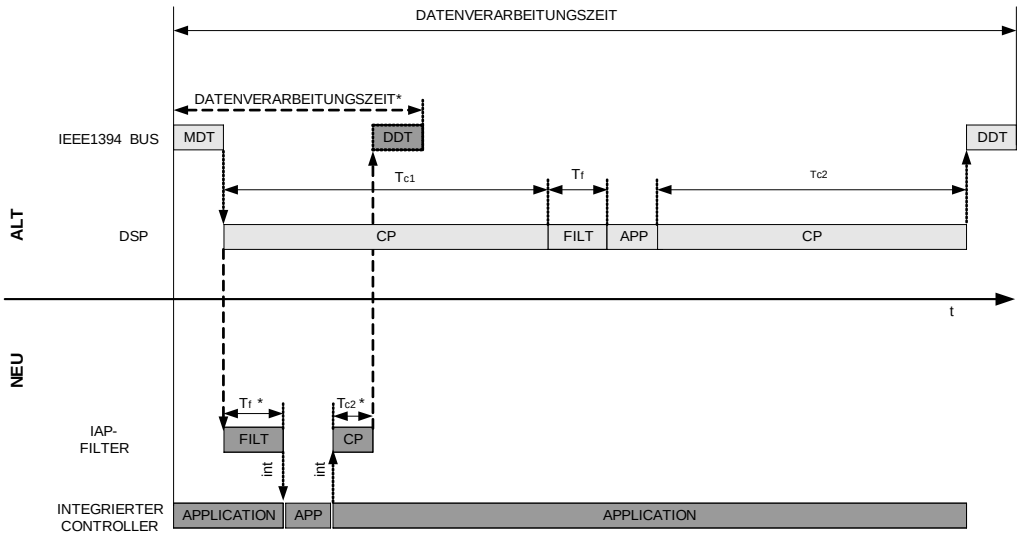



Abbildung 6.3: Vergleich der Datenverarbeitung in der aktuellen und der integrierten (zukünftigen) IAP-Knoten-Hardware (Zeitangaben für 20 Bytes Datengröße)

Aktoren bzw. Sensoren.

Der IAP-Filter hat dabei die Aufgabe, den über den IEEE-1394-Bus ankommenden Datenstrom gemäß konfigurierbarer Kriterien zu untersuchen und für die lokale Applikation relevante Daten weiter zu transportieren. Alle weiteren Softwareanteile des IEEE-1394-Treibers, des IAP, der Applikation und des Sensors können allesamt auf einem integrierten Controller ablaufen. Es ist weiter zu überlegen, inwiefern auch entsprechende schnelle Schnittstellen für Softwareteile aus der Applikation (z.B. die Filterung und Vorauswertung von Sensorsignalen) über das FPGA zur Verfügung gestellt werden können.

Abbildung 6.3 zeigt eine Gegenüberstellung der derzeitigen und der geplanten Aktivitäten innerhalb der Kommunikationsschicht im Zusammenhang mit der jeweils verwendeten Hardware. Nach dem Empfang des MDT arbeitet der DSP in der aktuellen Hardwarestruktur nach dem folgenden Sequenzablauf:

**CP** Empfangene Daten werden vom LLC-FIFO ins RAM kopiert. Dabei entspricht $T_{c1}$ dem im Abschnitt 4.4.2 verwendeten `T_Rx_DSP`

**FILT** Filterung der empfangenen Daten für die lokale Applikation

**APP** Ausführung der anwenderspezifischen Aufgaben. Dabei ist die Ausführungszeit im jeweiligen Knoten applikationsabhängig

**CP** Istwerte werden zurück in das LLC-FIFO kopiert und anschließend vom LLC über den Bus verschickt. Dabei entspricht $T_{c2}$ dem `T_Tx_DSP`

Durch den integrierten Ansatz wird die Verarbeitung in der Hardware wesentlich beschleunigt. Der IAP-Filter übernimmt die Interrupt-Auswertung sowie die Kopier-

und Filteraufgaben und sorgt somit für eine Entlastung der integrierten Recheneinheit, deren Kapazitäten auf diese Weise komplett für anwenderspezifische Aufgaben zur Verfügung stehen.

Schon durch den Einsatz des IAP-Filters mit einer FPGA-Taktfrequenz von $10MHz$ reduziert sich der Zeitaufwand für den Datenversand und -empfang bei einer Datengröße von $20Byte$ im IAP-Knoten auf insgesamt ca. $7\mu s$ ($T_f^* = 5\mu s, T_C^* = 2\mu s$). Die Filterungszeit ist dabei die Summe aus Kopierzeit und einem geschätzten $3\mu s$-Overhead. Durch diese Integration wird die Datenverarbeitung in den IAP-Knoten etwa um eine Größenordnung schneller. Bei einer FPGA-Implementierung der Funktionen kann von wesentlich höheren internen Taktfrequenzen ausgegangen werden ($50 - 100MHz$), so dass die in der derzeitigen Implementierung den Zyklus bestimmenden Latenzzeiten im Vergleich zum Regelungstakt vernachlässigbar werden. Dies schafft ein Optimierungspotential für den übergeordneten Regelzyklus.

Zusammengefasst bietet die Integration der Hardware folgende kommunikationstechnische Vorteile:

- Schnellerer Datenzugriff
- Reduzierte Datenverarbeitungszeit im IAP-Knoten
- Erhöhung der Reglerfrequenz im übergeordneten Steuerungssystem aufgrund der geringeren Datenverarbeitungszeit in den IAP-Knoten.
- Mehr Bandbreite für Ausführung anwenderspezifischer Aufgaben in den IAP-Knoten.

Außerdem wird durch die Integration eine Bauraum- und Gewichtsreduzierung der Kommunikationsknoten sowie Verkabelungseinsparung und bessere EMV-Eigenschaften durch eine geringere Anzahl externer Elemente und Signale erreicht. Dies bedeutet insgesamt eine Steigerung der Attraktivität der ganzen Kommunikations-Infrastruktur für industrielle Anwendungen.

Zusätzlich zu der konsequenten Weiterentwicklung der auf dem IEEE1394-Standard basierenden Kommunikationsschnittstelle wird das IAP auch im Hinblick auf die Anwendung von Industrial Ethernet weiter entwickelt. Ethernet gewinnt im industriellen Einsatz immer größere Bedeutung (auch im Bereich der Motion-Control-Anwendungen) und so ist zu erwarten, dass durch eine offene Ausrichtung des IAP auch die Integrationsmöglichkeit der SFB-Ergebnisse in industrielle Anwendungen begünstigt werden wird.

# Abbildungsverzeichnis

# Glossar

**Adressraum** Eine über eindeutige Adressen zugreifbare Menge von Speicherzellen. Man unterscheidet zwischen einem <physischen Adressraum> und einem <virtuellen Adressraum>.

**Anwendungsprofil** Das Anwendungsprofil eines Kommunikationsprotokolls legt fest, welche Datenstrukturen, Kommunikationsmechanismen und welches Verhalten beim Datenaustausch für eine spezielle Anwenderaufgabe notwendig ist. Beim ProfiBus-Kommunikationssystem spezifiziert „DP“ beispielsweise die realisierten Funktionalitäten, um dezentrale Peripherie in das Steuerungssystem einzubinden. „PA“ dagegen ermöglicht die Bedienung von Anwendungen der Prozessautomation.

**Application Programming Interface (API)** Siehe <Programmierschnittstelle>.

**Asynchrone Unterbrechung** Ein externes Ereignis, das die aktuelle Programmausführung unterbricht.

**Client** Ein Client ist Teil eines <Client/Server-Systems>. Ein Client wendet sich aktiv mit einem Auftrag / einer Anfrage an einen <Server>

**Client/Server-System** Organisationsform für Systeme. In einer Client/Server-basierten <Systemsoftware> wird der überwiegende Teil des Funktionsangebots durch deedizierte <Server> zur Verfügung gestellt, die von <Clients> und Servern selbst in Anspruch genommen werden. Die Adressraum-übergreifende Kommunikation zwischen Client und Server wird in diesem Fall von einem <Mikrokern> realisiert.

**Condition-Variable** Prozesse warten an einer Condition-Variable innerhalb eines <Monitors> auf den Eintritt einer bestimmten Bedingung. Andere Prozesse, die zwischenzeitlich im Besitz eines Monitors sind, können den Eintritt einer bestimmten Bedingung signalisieren und damit evtl. blockierte Prozesse befreien.

**Deadline** Siehe <Frist>

**Deadlock** Siehe <Verklemmung>

**Dispatcher** Softwarefunktionalität, die je nach Kontext eine spezifische Funktions-“Abfertigung“ realisiert. Im Bereich der Betriebssysteme realisiert der Dispatcher die Übergänge im <Zustandsmodell>. Er führt dabei alle Funktionalitäten aus, die bei einem <Kontextwechsel> zwischen zwei <Threads> notwendig

sind. Zur Realisierung entfernter Methodenaufrufe (<Remote Methode Invocation>) innerhalb verteilter Systeme sorgt der Dispatcher für den Aufruf der vom <Proxy> gewünschten Methode innerhalb des entfernten <Prozesses>.

**Dynamischer Datenspeicher** In der Größe veränderlicher Bereich, in dem die von einem <Thread> dynamisch zur Laufzeit angeforderten Speicherbereiche verwaltet werden.

**Echtzeitsystem** Anwendung, bei der Zeitvorgaben eingehalten werden müssen. Man unterscheidet zwischen <strikten Echtzeitsystemen> und <schwachen Echtzeitsystemen>

**Fieldbus Message Specification (FMS)** Spezifikation für Feldbus-Nachrichten zur objektorientierten Darstellung von Prozessobjekten auf Kommunikationsobjekte mit dem Ziel der Verbindung verteilter Anwendungsprozesse zu einem Gesamtprozess. Dabei erfolgt auch eine Berücksichtigung der Anbindung an hierarchisch übergeordnete Systeme.

**Frist** Zeitpunkt bis zu dem eine Echtzeitaktivität beendet sein muss.

**Gemeinsamer Speicherbereich** Mehrere <Threads> verfügen über einen teilweise oder vollständig gemeinsamen Speicherbereich. Alle beteiligten Threads können über Lese- und Schreiboperationen darauf zugreifen.

**Geräteprofil** Ein Geräteprofil beschreibt die Funktionalitäten und den Aufbau eines Objektverzeichnisses für eine bestimmte Klasse von Geräten, die in einer Steuerungsanwendung bedient werden müssen. Beispielsweise existieren Geräteprofile für IO-Module, elektrische Antriebe, Sensoren,... Darin werden Wertebereiche, Auflösungen und Einheiten definiert. Die Existenz von Geräteprofilen bewirkt eine höhere Unabhängigkeit der Steuerungshersteller von den Geräteherstellern, sofern sich diese an die Definitionen halten.

**Gleichzeitigkeit** Ein Maß für den zeitlichen Versatz der Ausführung einer Funktionalität in mehreren Teilnehmern (Geräten). Zur Bestimmung der Echtzeitfähigkeit bestimmt dieses Kriterium gemeinsam mit der <Synchronität> die Möglichkeiten des synchonen Ausführens koordinierter Aktivitäten.

**Hub** Eine kostengünstige Lösung für die Erweiterung von Informationsnetzwerken. Einkommende Datentelegramme werden durch die Verarbeitung auf den ISO/OSI-Schichten 1 und 2 auf alle übrigen Ports dupliziert. Es erfolgt hierbei keine Filterung.

**Interrupt** Ein externes Ereignis, das die aktuelle Programmausführung unterbricht (Asynchrone Unterbrechung).

**Kontextwechsel** Wechsel in einer <Thread>-Ausführung. Der Zustand des aktuell ausgeführten Threads wird im <PCB> gespeichert. Anschließend wird der Zustand des nächsten auszuführenden Threads wieder hergestellt.

**Laufzeitmodell** Ein Laufzeitmodell ist eine Modellvariante, die durch eine bestimmte Funktionsmenge und Systemarchitektur beschrieben wird. Für ein Laufzeitmodell existieren verschiedene Ausprägungen, die sich z.B. bezüglich der enthaltenen Systemdienste und der jeweiligen Spezialisierung unterscheiden.

**Laufzeitplattform** Die Funktionsmenge der <Systemsoftware> bildet die Laufzeitplattform. Die Laufzeitplattform basiert auf einem bestimmten <Laufzeitmodell>.

**Laufzeit-Stapelspeicher** Ein Laufzeit-Stapelspeicher speichert die Aufrufverschachtelung und die damit verbundenen lokalen Variablenzustände eines <Threads>. Je nach Prozessortyp können Laufzeit-Stapelspeicher von hohen zu niedrigen oder von niedrigen zu hohen Adressen wachsen.

**Marshaling** Funktionsablauf zur rechnerhardware-abhängigen Übertragung strukturierter Daten in eine einheitliche Datenrepräsentation (z.B. einen seriellen Datenstrom) zur Übertragung. Einfachstes Beispiel ist die Anwendung des <Marshaling> und <Unmarshaling> bei der Anpassung von zwei miteinander kommunizierenden Recheneinheiten, die ihre Daten jeweils im Big-Endian- bzw. Little-Endian-Datenformat repräsentieren.

**Memory Management Unit (MMU)** Hardwarekomponente für die effiziente Umsetzung virtueller Speichertechniken. Die MMU kann als eigener Chip realisiert oder auf dem Prozessorchip platziert sein.

**Modul** Im Bereich der Programmierung ein Kontainer für Quellcode, der in seinem Modul-Status nicht mit anderen Prozessor-Komponenten interagieren kann. Wird das Modul kompiliert und gestartet, so entsteht aus ihm ein <Thread>, der im zugehörigen Prozess-Kontext lauffähig ist und zur Laufzeit mit anderen Threads (im eigenen oder in anderen Prozessen) interagieren kann.

**Monitor** Ein sprachbasiertes Synchronisationsmittel, bei dem die gemeinsam genutzten Daten und die darauf definierten Zugriffsfunktionen zu einer Einheit zusammengefasst werden. Die Monitorfunktionen schließen sich in der Ausführung wechselseitig aus.

**Nichtpräemptives Scheduling** Ein Schedulingverfahren, bei dem einem rechnenden <Thread> der zugeordnete Prozessor nur entzogen werden kann, wenn eine blockierende Operation aufgerufen oder der Prozessor freiwillig abgegeben wird (yielding).

**PCB** Siehe <Prozess-Kontrollblock>

**Physischer Adressraum** Die über den Adressbus des Prozessors direkt zugreifbare Speicher- und E/A-Hardware. Eine eventuell vorhandene <Memory Management Unit> ist in diesem Fall ausgeschaltet.

**POSIX** Ein auf UNIX basierender Standard zur Gewährleistung von Software-Portabilität auf Quellcode-Ebene. Wenn ein Betriebssystem POSIX-konform ist, so sollte ein für es geschriebenes C-Programm auf anderen POSIX-komformen Betriebssystemen sofort lauffähig sein, sofern nur POSIX-konforme Funktionsaufrufe verwendet wurden. Der POSIX-Standard wird von der Organisation IEEE entwickelt und von den Organisationen ANSI und ISO standardisiert. POSIX 1003.1b (früher POSIX.4 bezeichnet) beinhaltet Definitionen für die Echtzeit-Anwendung.

**Präemptives Scheduling** Ein Schedulingverfahren, bei dem einem rechnenden <Thread> der zugeordnete Prozessor entzogen werden kann, wenn eine blockierende Operation aufgerufen, der Prozessor freiwillig abgegeben (yielding) oder durch eine <asynchrone Unterbrechung> ein zweiter Prozess mit höherer Priorität rechenbereit wird. Ein typisches Beispiel für eine Präemption ist ein Thread-Wechsel aufgrund eines Timer-Interrupts (z.B. Round-Robin-Scheduling).

**Priorität** Kennzeichnung eines <Thread> nach seiner Wichtigkeit. Die Priorität bestimmt die Schedulingreihenfolge bei mehreren rechenbereiten Threads.

**Profil** Ein Standard für ein Kommunikationsprotokoll bedient einen großen Bereich von möglichen Anwendungen in einer industriellen Umgebung, von der Steuerungs- und Leitebene bis hinunter zur Feldebene. Um den in der Regel umfangreichen Funktionsumfang dieses Kommunikationsprotokolls zu ordnen und dabei einen benötigten Funktionsumfang für eine Klasse von Anwendungen oder Geräte zu definieren, werden oft zusätzlich zum Protokoll spezielle <Anwendungsprofile> bzw. <Geräteprofile> definiert.

**Programmierschnittstelle** Eine Beschreibung der Funktionsmenge eines Dienstes, die von einem Programmierer bei der Nutzung dieses Dienstes in Anspruch genommen werden kann. Die Programmierschnittstelle liegt in einer von der jeweiligen Programmiersprache abhängigen Form vor, in der alle Funktionen durch Angabe von Namen und Signatur sowie die Datentypen für Argumente und Rückgabewerte eindeutig definiert werden.

**Proxy** Softwarefunktionalität, die eine stellvertretende Funktion für einen Aufrufer ausführt, um beispielsweise eine Aufruftransparenz zu erzielen. Bei der Anwendung von entfernten Methodenaufrufen (<Remote Methode Invocation>) innerhalb verteilter Systeme nimmt sie den Methodenaufruf entgegen und übersetzt ihn in einen notwendigen Kommunikationsvorgang zum entfernten Prozess/Objekt. Dazu erfolgt das <Marshaling> der Methodenargumente, das Versenden, das Warten auf die Antwort und deren Rückgabe zurück an den Aufrufer. Die internen Vorgänge bleiben für den Aufrufer unsichtbar.

**Prozess-Kontrollblock** Datenstruktur, die alle relevanten Informationen zur Verwaltung von <Prozessen> und <Threads> speichert. Auf den PCB greifen im wesentlichen <Dispatcher> und <Scheduler> zu.

**Remote Methode Invocation, RMI** ...

**Remote Procedure Call, RPC** Der Adressraumübergreifende Aufruf einer Prozedur. Gängiger Vertreter der auftragsbasierten Nachrichtenkommunikation in einem <Client/Server-System>

**Repeater** Repeater dienen innerhalb eines Netzwerks dazu, die maximale Ausdehnung zu erweitern. Einige Repeater sind in der Lage, unterschiedliche physikalische Medien in einem Netzwerk miteinander zu verbinden. Dabei erfolgt die Verarbeitung der Datentelegramme nur auf ISO/OSI-Schicht 1.

**Router** Router kommen zum Einsatz, um innerhalb eines Netzwerks Daten für unterschiedliche Anwendungsbereiche zu trennen. Es sind dabei die ISO/OSI-

Schichten 1 bis 4 bei der Auswertung der Telegramme beteiligt, die je nach Filterung auf unterschiedliche Ausgangsports repliziert werden.

**Scheduler** Teil der <Systemsoftware>, der aus einer Menge rechenwilliger <Threads> den nächsten auszuführenden Kandidaten auf der Grundlage einer konkreten Schedulingstrategie auswählt. Die Notwendigkeit einer Auswahl ist immer bei einem Assign-Übergang durch den <Dispatcher> gegeben.

**Schwaches Echtzeitsystem** Echtzeitsystem, bei dem die Verletzung einer Echtzeit-Vorgabe (>Frist>) keine schwerwiegenden Folgen hat. Trotzdem ist eine Fristverletzung nach Möglichkeit zu vermeiden.

**Server** Ein Server ist Teil eines <Client/Server-Systems>. Ein Server wartet passiv auf eintreffende Aufträge eines <Clients>. Nach Auftragsbearbeitung wird das Auftragsergebnis in Form eines Reply an den Client zurückgesendet.

**Shared Memory** Siehe <Gemeinsamer Speicherbereich>

**Skeleton** Softwarefunktionalität, die zur Realisierung entfernter Methodenaufrufe innerhalb verteilter Systeme dafür sorgt, dass das <Unmarshaling> der übergebenen Methodenargumente erfolgt und die erwünschte Methode des entfernten Prozesses/Objekts ausgeführt wird. Das Skeleton beantwortet den entfernten Methodenaufruf nach Ausführung der Methode und <Marshaling> der Antwortdaten an den aufrufenden <Proxy>.

**Stack** Siehe <Laufzeit-Stapelspeicher>

**Striktes Echtzeitsystem** Ein Echtzeitsystem, bei dem die Verletzung einer Zeitvorgabe (<Frist>) katastrophale Folgen haben kann.

**Switch** Über einen Switch werden Informationsnetzwerke in der Weise erweitert, dass einkommende Datentelegramme anhand ihrer Zieladresse gefiltert und auf den jeweiligen Port ausgegeben werden, an dessen Ende der Adressat erwartet wird. Es erfolgt also eine Verarbeitung auf den ISO/OSI-Schichten 1 bis 3, und damit eine Reduktion der Kollisionswahrscheinlichkeit bei CSMA/CD Netzwerken. Ein Switch „lernt“ die Netzwerktopologie durch Auswertung der Senderadressen der einzelnen Telegramme.

**Synchronität** Im Bereich der Antriebstechnik nicht klar definiertes zeitliches Verhältnis. Es bezieht sich allgemein auf den zeitlichen Jitter bei der Abarbeitung immer der gleichen lokalen Funktionalität. Zur Bewertung von Echtzeitkriterien sollte auch die <Gleichzeitigkeit> berücksichtigt werden.

**Systemsoftware** Konfigurierbare und erweiterbare Menge von Systemdiensten, die einer Anwendung die einfache und von technischen Details befreite Nutzung eines Rechnersystems ermöglicht. Die Funktionalität der Systemsoftware basiert auf einem bestimmten <Laufzeitmodell> und steht in der Anwendung in Form einer <Laufzeitplattform> zur Verfügung.

**Thread** Abstraktion eines physischen Prozessors. In dieser Rolle ist der Thread Träger einer sequentiellen Aktivität, die durch die Ausführung einer ihm zugeordneten Instruktionsfolge (Programm) bestimmt ist. Die Ausführung findet im

Kontext des <Adressraums> des zugehörigen <Prozesses> statt. Threads innerhalb eines Prozesses sind nicht gegeneinander abgeschirmt. Einem Thread wird durch den <Dispatcher> zeitweise oder dauerhaft ein physischer Prozessor zugeordnet.

**Trajektorie** Bewegungs-Koordinaten (Bahn) eines Körpers im Raum, die als Funktion der Zeit angegeben sind. Es werden also nicht nur die absoluten (oder relativen) Koordinaten eines Körpers, sondern auch der genaue Zeitpunkt für diese Koordinaten berücksichtigt.

**Unmarshaling** Funktionsablauf zur rechnerhardware-abhängigen Rück-Übertragung aus einer einheitlichen Datenrepräsentation (z.B. ein serieller Datenstrom) in strukturierte Daten nach einer erfolgten Datenübertragung in einem verteilten System.

**Verklemmung** Eine zirkuläre Wartebedingung zwischen mehreren <Threads>. Die betroffenen Threads können sich aus einer Verklemmung nicht selbständig befreien.

**Virtueller Adressraum** Abstraktion des <physikalischen Adressraums>. Dabei können viele technische Beschränkungen wie z.B. die maximale Größe eines Adressraums praktisch aufgehoben werden. Durch virtuelle Adressräume können auch Schutzkonzepte verwirklicht und insbesondere Anwendungen isoliert werden. Zum Einsatz kommen segment- und seitenbasierte Verfahren, die mit Hilfe der <Memory Management Unit> umgesetzt werden.

# Literaturverzeichnis


[1] Wang, S.: *Specification, Allocation and Schedulability Analysis for Fixed-Priority Hard Real-Time Systems*. Dissertation. Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, 1999

[2] Hruschka, P.; Rupp, C.: *Agile Softwareentwicklung für Embedded Real-Time Systeme mit der UML*. Hanser-Verlag, 2002

[3] Borg...

[4] Nehmer, J.; Sturm, P.: *Systemsoftware - Grundlagen moderner Betriebssysteme*. 2. aktualisierte Auflage, dpunkt verlag, Heidelberg, 2001

[5] Tanenbaum, A.: *Modern Operating Systems*. 2nd editioni, Prentice-Hall, 2001

[6] Brause, R.: *Betriebssysteme - Grundlagen und Konzepte*. 2. überarbeitete Auflage, Springer, 2001

[7] QNX Software Systems Ltd.: *QNX Realtime Operating System - System Architecture*. Third edition, Canada, 2001

[8] Holt, R. C.: *Some Deadlock Properties of Computer Systems*. ACM Computing Surveys, Vol. 4, Nr. 3, 1972, pp. 179-196

[9] Silberschatz, A.; Gagne, G.; Galvin, P.B.: *Operating System Concepts*. John Wiley and Sons. inc, 7th ed., US, 2005

[10] Coffmann, E. G.; Elphick, M. J.; Shoshani, A.: *System Deadlocks*. ACM Computing Surveys, Vol. 3, Nr. 2, 1971, pp. 67-78

[11] Habermann, A. N.: *Prevention of System Deadlocks*. CACM, Vol. 12, 1969, pp. 373-377

[12] Stallings, W.: *Betriebssysteme - Prinzipien und Umsetzung*. Autorisierte Übersetzung (2003) des Buches: Operating Systems - Internals and Design Principles. 4th edition, Prentice Hall, 2001

[13] Solomon, D.: *The Windows NT Kernel Architecture*. Computer, Oktober, 1998

[14] *Hard realtime control and Windows NT*. Article, in: Industrial Networking and Open Control, October, 2003

[15] *Die neuen Embedded Versionen*. Artikel, Zeitschrift, Computer & Automation, Juni, 2001

[16] *Der Nachfolger von CE 3.0.* Artikel, Zeitschrift, Computer & Automation, Februar, 2002

[17] *Sportliche Tuning-Kits - Zusatzprodukte machen Windows XP Embedded echtzeitfähig.* Artikel, Zeitschrift, Elektronik, 16/2003

[18] Rusling, D. A.: *The Linux Kernel.* http://metalab.unc.edu/LDP/LDP/tlk/tlk.html, Jan. 1999

[19] www.venturcom.com

[20] Knoll, A.: *Echtzeitsysteme - Echtzeitbetriebssysteme.* Vorlesungsskript, Lehrstuhl Embedded Systems and Robotics, TU München, SS, 2005

[21] G. Beckmann: *Ein Hochgeschwindigkeits-Kommunikations-System für die industrielle Automation.* Dissertation, Technische Universität Braunschweig, 2001

[22] Beckmann, G.; Varchmin, J.-U.: *Industrial Automation Protocol Version 0.9.4.* Spezifikation eines Kommunikationsprotokolls für die industrielle Automation auf Basis des IEEE1394, Institut für Elektrische Messtechnik und Grundlage der Elektrotechnik der TU Braunschweig, Braunschweig, 2001.

[23] Balen, H.: *Distributed Object Architectures with CORBA.* Cambridge, 2000

[24] Object Management Group, http://www.omg.org, 2000

[25] Bormann, A.; Hilgenkamp, I.: *Industrielle Netze - Ethernet-Kommunikation für Automatisierungsanwendungen.* Hüthig-Verlag, Heidelberg, 2006

[26] Müller, M.: *PROFInet - das Bussystem der Zukunft - die strategische Ausrichtung der Automatisierungstechnik.* Interview, Zeitschrift, Automatisierungstechnische Praxis (atp), September, 2005

[27] Schnell, G.; Wiedemann, B.: *Bussysteme in der Automatisierungs- und Prozesstechnik.* Vieweg-Verlag, Wiesbaden, 2006

[28] Breyer, R.; Riley, S.: *Switched, Fast, and Gigabit Ethernet.* Third edition, Macmillan Technical Publishing, USA, 2002

[29] Furrer, F.J.: *Ethernet-TCP/IP für die Industrieautomation.* Grundlagen und Praxis, 2.Auflage, Hüthig, Heidelberg, 2000

[30] Bziuk, W.: *Digitale Nachrichtenvermittlung.* Vorlesungsskript, Institut für Datentechnik und Kommunikationsnetze, TU Braunschweig, 2006

[31] Institute for Electrical and Electronics Engineers, IEEE - http://www.ieee.org

[32] Internet Society, ISOC - http://www.isoc.org

[33] Internet Engineering Task Force, IETF - http://www.ietf.org

[34] Internet Architecture Board, IAB - http://www.iab.org

[35] Internet Assigned Numbers Authority, IANA - http://www.iana.org

[36] Internet Corporation for Assigned Names and Numbers, ICANN - http://www.icann.org

[37] Internet Research Task Force, IRTF - http://www.irtf.org

[38] Computer Emergency Response Team, CERT - http://www.cert.org

[39] Deutsches Network Information Center, DeNIC eG - http://www.denic.de

[40] Interbus Club - http://www.interbusclub.com

[41] Internet Network Information Center, InterNIC eG - http://www.internic.net

[42] World Wide Web Consortium, W3C - http://www.w3.org

[43] IAONA - http://www.iaona-eu.com

[44] Profibus Nutzer Organisation, PNO - http://www.profibus.com

[45] IEEE Std 1588-2002: *IEEE Standard for a Precision Clock Synchronisation Protocol for Networked Measurement and Control Systems*

[46] IEEE Std 802.3ae-2002: *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications; Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Measurement Parameters for 10 GB/s Operation*

[47] Jasperneite, J.: *Echtzeit-Ethernet im Überblick.* Artikel, ATP - Automatisierungstechnische Praxis, Heft 3, 2005

[48] Felser, M.: *Real-Time Ethernet - Industry Prospective.* Article, Proceedings of the IEEE, Vol. 93, No. 6, June, 2005

[49] International Electrotechnical Commission, IEC 61784-2, Digital data communications for measurement and control - Part 2: *Additional profiles for ISO/IEC 8802-3 based communication networks in real-time applications.*

[50] Schwager, J.: *Ethernet erreicht das Feld - Teil 1.* Artikel, Elektronik, Nr. 11, 2004

[51] Schwager, J.: *Ethernet erreicht das Feld - Teil 2.* Artikel, Elektronik, Nr. 13, 2004

[52] Chiueh, T.: *RETHER: A Software-Only Real-Time Ethernet for PLC Networks.* Proceedings of the Embedded Systems Workshop, Cambridge, Massachusetts, USA, 1999

[53] Modbus/IDA Organisation - http://www.modbus.org

[54] IEC: *Real time ethernet modbus-RTPS, Proposal for a Publicly Available Specification for Real-Time Ethernet.* Document, IEC 65C/341/NP

[55] Fielabus Foundation - http://www.fieldbus.org

[56] EtherCAT Technology Group - http://www.ethercat.org

[57] Janssen, D.: *EtherCAT als Antriebsbus.* Artikel, ATP - Automatisierungstechnische Praxis, Heft 2, 2005

[58] PROFINET Technology and Application, System Description - http://www.profibus.com/pall/meta/downloads/article/00456

[59] Pigan, R.; Metter, M.: *Automatisieren mit PROFINET - Industrielle Kommunikation auf Basis von Ethernet.* Siemens AG, 2005

[60] Industrial Ethernet Book - http://ethernet.industrial-networking.com/articles

[61] Popp, M.: *Das PROFINET IO-Buch.* Hüthig, Heidelberg, 2005

[62] Ethernet Industrial Protocol (Ethernet/IP) - http://www.ethernetip.de

[63] Ethernet Powerlink Standardization Group (EPSG) - http://www.ethernet-powerlink.org

[64] Heyer-Reinfeld, A.: *Ethernet Powerlink - Auf Tuchfühlung zu gängigen Standards.* Zeitschriftenartikel, IEE, Antriebstechnik, Nr.11, 2003

[65] Dreher, A.: *Ethernet in der industriellen Datenkommunikation - Wie viel Zeit braucht Echtzeit.* Zeitschriftenartikel, IEE, Datentechnik, Nr.03, 2003

[66] SERCOS interface interest group - http://www.sercos.org

[67] Jetter AG - www.jetter.de

[68] Anderson, D.: *FireWire System Architecture.* PC System Architecture Series, second edition, Addison Wesley, 2001

[69] Scholles, M.: *IEEE 1394 für die Automation.* Artikel. Computer & Automation, Mai 2005

[70] Scholles, M.: *Bündelung der 1394-Kräfte.* Markt und Technik Sonderheft, 2005

[71] Frommhagen, K.; Nauber, P.; Schellinski, U.; Scholles, M.: *1394AP - A Protocol for deterministic Industrial Communication via IEEE 1394.* 10th IEEE conference on Emerging Technologies and Factory Automation, ETFA, 2005

[72] MINDREADY - www.mindready.com

[73] ORSYS - www.orsys.de

[74] 1394TA - www.1394ta.org/Technology/About/Renault_IDB1394_2.mpg

[75] IEEE Std 1394-1995: *Standard for a High Performance Serial Bus*

[76] IEEE Std 1394a-2000: *Standard for a High Performance Serial Bus - Amendment 1*

[77] IEEE Std 1394b-2002: *Standard for a High Performance Serial Bus - Amendment 2*

[78] IEEE Std 1394c-200x: *Draft Standard for a High Performance Serial Bus - Amendment 3*

[79] European Cooperation for Space Standardization: *SpaceWire - links, nodes, routers and networks.* ECSS-E-50-12A, January, 2003

[80] IEEE Computer Society: *IEEE Standard for Heterogeneous Interconnect (HIC) (Low-Cost,Low-Latency ScalableSerial Interconnect for ParallelSystem Construction)*. IEEE Standard 1355-1995, IEEE, June 1996

[81] Telecommunications IndustryAssociation: *Electrical Characteristics of Low Voltage Differential Signaling (LVDS) Interface Circuits*. ANSI/TIA/EIA-644, March 1996

[82] IEEE Computer Society: *IEEE Standard for Low-Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI)*. IEEE Standard 1596.3-1996, IEEE, July 1996.

[83] STAR-Dundee - www.star-dundee.com

[84] SSD Drives Homepage, http://www.ssddrives.de/890.htm, 2004

[85] Mehrere Achsen Steuerung, http://www.all-electronics.de/product/45ff41b53a6.html, Juli 2005

[86] Grundmann, U.: *Java betritt das Parkett*. Artikel, Zeitschrift, Computer & Automation, Januar, 2002

[87] Chung, P.E.; Huang, Y.; Yajnik, S.; Liang, D.; Shih, J.; Wang, C.; Wang, Y: *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer*. Webdokument, http://www.bell-labs.com/emerald/dcom-corba/Paper.html, 2001

[88] The Open Group: *Distributed Computing Environment*. Webdokument, http://www.camb.opengroup.org/tech/dec, 2001

[89] Jia, S.; Takase, K.: *Internet-Based Robotic System Using CORBA as Communication Architecture*. Journal of Intelligent and Robotic Systems, Robotik 2002, Ludwigsburg, Germany, pp. 225-230, 2002

[90] Schmidt, D.C.; Levine, D.L.; Mungee, S.: *The Design of the Tao Real-time Object Request Broker*. Computer Communications, Elsivier Science, Vol.21, pp.294-324, No.4, 1998

[91] Hasegawa, T.; Suehiro, T.; Takase, K.: *A Model-Based Manipulation System with Skill-Based Execution*. IEEE Transactions on Systems, Man and Cybernetics, 1981, pp. 535-533

[92] Mason, M.T.: *Compliance and Force Control for Computer Controlled Manipulators*. IEEE Transactions on Systems, Man and Cybernetics, Juni 1981

[93] Gosselin, C. M.: *Determination of the Workspace of a 6 - DOF Parallel Manipulator*. Journal of Mech. Design 112 (1990) 3, 1990, pp 331-333

[94] Gosselin, C. M.; Angeles, J.: *Singularity Analysis of Closed - Loop Kinematic Chains*. IEEE - Transactions On Robotics and Automation 6 (1990) 3, 1990, pp 281-290

[95] Kumar, V.: *Characterization of Workspaces of Parallel Manipulators*. Trans. ASME: Journal of Mechanical Design 114, 1992, pp 349-358

[96] Bruyninckx, H.; deShutter, J.: *Specification of Force-Controlled Actions in the Task Frame Formalism - A Synthesis*. IEEE Transactions on Robotics and Automation, August 1996

[97] Thomas, U.; Maaß, J.; Wahl, F.M.; Hesselbach, J.: *Towards a New Concept of Robot Programming in High Speed Assembly Applications*. IEEE International Conference on Intelligent Robots and Systems, 2005

[98] Kröger, T.; Finkemeyer, B.; Wahl, F.M.: *A Task Frame Formalism for Practical Implementations*. IEEE International Conference on Robotics and Automation, pp.5218-5223, 2004

[99] Siciliano, B.; Villani, L.; Caccavale, F.: *Kinematics, Dynamics and Control for a Class of Parallel Robots*. 2.Internationales Kolloquium des Sonderforschungsbereichs 562, Braunschweig, 2005

[100] Bier, C.: *Geometrische und physikalische Analyse von Singularitäten bei Parallelstrukturen*. Dissertation, Technische Universität Braunschweig, 2006

[101] World Wide Web Consortium: *W3C Recommendation: Extensible Markup Language (XML) v1.1*. 2004, http://www.w3.org/TR/2004/REC-xml11-20040204/

[102] World Wide Web Consortium: *W3C Recommendation: Simple Object Access Protocol (SOAP) v1.2*. 2003, http://www.w3.org/TR/soap/

[103] Tanenbaum, A.S.: *Computer Networks*. 4. Auflage, Prentice-Hall, 2003 [Cou2001] Coulouris, G.; Dollimore, J.; Kindberg, T.: Distributed Systems: Concepts and Design. 3rd edition, Addison-Wesley, 2001

[104] Coulouris, G.; Dollimore, J.; Kindberg, T.: *Verteilte Systeme - Konzept und Design*. 3.Auflage, Pearson-Verlag, 2002

[105] Schmidt, D. C.; O'Ryan, C.: *Pattern and Performance of Distributed Real-time and Embedded Publisher/Subscriber Architectures*. Department of Electrical & Computer Engineering, University of California, Irvine, 2002

[106] Schantz, R. E.; Loyall, J. P.; Rodrigues, C.; Schmidt, D. C.; Krishnamuthy, Y.; Pyarali, I.: *Flexible and Adaptative QoS Control for Distributed Real-Time and Embedded Middleware*. 2003

[107] Österle, H.; Riehm, R.; Vogler, P.: *Middleware - Grundlagen, Produkte und Anwendungsbeispiele für die Integration heterogener Welten*. Vieweg Verlag, 1996

[108] Corsaro, A.; Schmidt, D.C.: *The Design and Performance of Real-Time Java Middleware*. IEEE Transactions on parallel and distributed systems, vol. 14, no. 11, November 2003

[109] Arulanthu, A. B.; O'Ryan, C.; Schmidt, D. C.; Kircher, M.; Parsons, J.: *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging*. Department of Computer Science, Washington University, St. Louis, MO 63130, 2000

[110] Schmidt, D. C.; Kuhns, F.: *An overview of the real-time CORBA specification*. Computer, 33, 2000

[111] Schmidt, D. C.; Deshpande, M.; O'Ryan, C.: *Operating System Performance in Support of real-time Middleware*. In 7th IEEE International Workshop on object-oriented real-time dependable Systems, Januar, 2002

[112] Orfali, R.; Harkey, D.; Edwards, J.: *Instant CORBA*. Addison-Wesley, 1998

[113] Finkemeyer, B.; Borchard, M.; Wahl, F.M.: *Robot Control Architecture Based on an Object Server*. IASTED International Conference on Robotics and Manufacturing, Cancun, Mexico, 2001, pp. 36-40

[114] Finkemeyer, B.; Borchard, M.: *MIRPA: A Middleware for Open Robot Control Systems*. Robotik 2002, Ludwigsburg, Germany, pp. 225-230, 2002

[115] Finkemeyer, B.: *Robotersteuerungsarchitektur auf der Basis von Aktionsprimitiven*. Dissertation, TU Braunschweig, 2004

[116] Schulz, G.: *Performance Testing of the TENA Middleware*. OMG's Workshop on Distributed Object Computing for Real-Time and Embedded Systems, Arlington, VA, 2003

[117] Utz, H.; Sablatnög, S.; Enderle, S.; Kraetzschmar, G.: *Miro – Middleware for Mobile Robot Applications*. IEEE Transactions on Robotics and Automation, Vol. 18, No. 4, August 2002

[118] Lutz, P.; Sperling, W.: *OSACA - the vendor neutral control architecture*. In European Conference on Integrating in Manufactoring IiM, Dresden, 1997

[119] *Final Report - OSACA I + II*. Webdokument, http://www.osaca.org/osaca/-_pdf/os2fin4.pfd, 1996

[120] Sperling, W.; Lutz, P.: *Enabling Open Control Systems - An Introduction to the OSACA System Platform*. Robotics and Manufacturing, Vol. 6, ISRAM'97 Montpellier/France, ASME Press New York, 1996

[121] Modular Control Architecture 2 (MCA2). Kay-Ulrich Scholl. http://mca2.sourceforge.net/

[122] Schröder, J.; Steinhaus, P.; Gockel, T.; Dillmann, R.: *Design of a Holonomous Platform for a Humanoid Robot using MCA2*. Proceedings of the 8th Conference of Intelligent Autonomous Systems, Netherlands, 2004

[123] Diethers, K.; Finkemeyer, B.; Kohn, N.: *Middleware zur Realisierung offener Steuerungssoftware für hochdynamische Prozesse*. Artikel, Information Technology (it), Fachzeitschrift, Nr. 1, Seiten 39-47, 2004

[124] Diethers, K.; Kohn, N.; Kolbus, M.; Reisinger, T.; Steiner, J.; Thomas, U.: *PROSA - A Generic Control Architecture for Parallel Robots*. Vortrag, MECHROB 2004, Konferenz, Aachen, 2004

[125] Kohn, N.; Steiner, J.; Varchmin, J.-U.; Goltz, U.: *Universal Communication Architecture for High-Dynamic Robot Systems using QNX*. Vortrag, ICARCV 2004, Konferenz, Kunming, China, 2004

[126] Kohn, N.; Varchmin, J.-U.: *Kommunikations-Infrastruktur für Steuerungen in hochdynamischen Robotersystemen.* Vortrag und Tagungsband, DFMRS, Fachtagung Automatisierung, Bremen, 2004

[127] Kohn, N.; Varchmin, J.-U.: *Kommunikation (IEEE1394) und Middleware zur Steuerung hochdynamischer Prozesse.* Vortrag und Tagungsband, SPS/IPC/DRIVES, Automatisierungstechnik-Kongress, Nürnberg, 2003

[128] Kohn, N.; Beckmann, G.; Varchmin, J.-U.: *Architecture of a Realtime Communication Network for Parallel Robots.* Vortrag + Tagungsband, 1st International Colloquium, Collaborative Research Centre 562, Braunschweig, 2002

[129] Kohn, N.; Varchmin, J.-U.; Michalik, H.: *A New Communication Architecture for Hard Realtime Robot Control.* Vortrag + Tagungsband, 2nd International Colloquium, Collaborative Research Centre 562, Braunschweig, 2005

[130] Maaß, J.; Kohn, N.; Bruhn, M.; Hesselbach, J.: *Open Modular Robot Control Architecture for Assembly Using the Task Frame Formalism.* Artikel, International Journal of Advanced Robotic Systems, Vol. ???, pp. ???, 2005

[131] Sonderforschungsbereuch 562, Erstantrag: *Robotersysteme für Handhabung und Montage*, Deutsche Forschungsgemeinschaft (DFG), 2000

[132] Sonderforschungsbereich 562, Ergebnisbericht für den 1. Förderungszeitraum 2000-2003, Deutsche Forschungsgemeinschaft (DFG), 2003

[133] Sonderforschungsbereich 562, Fortsetzungsantrag für den 3. Förderungszeitraum 2006-2010, Deutsche Forschungsgemeinschaft (DFG), 2006

[134] Steiner, J.; Kohn, N.: *Industrial Automation Protocol - User and Developer Guide.* Software-Handbuch, Version 1.2r2, Institut für Elektrische Messtechnik und Grundlagen der Elektrotechnik, 2006

[135] Kohn, N.: *EMG IEEE1394 Treibersoftware - Benutzerhandbuch.* Software-Handbuch, Version 1.5r3, Institut für Elektrische Messtechnik und Grundlagen der Elektrotechnik, 2005

[136] Binder, D.: *Vorteile herstellerübergreifender Module offener Steuerungen.* Bey, I. (Hrsg.): *Produktion 2000 - Ergebnisse und Zukunftschancen.* Tagungsband Karlsruher Arbeitsgespräch 1998, Karlsruhe 1998, S. 233-253.

[137] Pritschow, G.: *Offene Steuerungen - ein Gebot der Stunde.* WT-Produktion und Management (1992), Nr. 11, S. 3.

[138] Pritschow, G.; Lutz, P.: *Offene Standards in der Steuerungstechnik.* Steuerungstechnisches Kolloquium, ISW Stuttgart, 1999.

[139] D'Souza D. F., Wills A. C.: *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1998.

[140] Honegger, M.: *Adaptive nichtlineare Steuerung und Regelung des Hexaglide.* 1. Internationales Parallelkinematik-Kolloquium, Zürich, Schweiz, 1998, S. 33-41.

[141] Beji L. et al.: *Non Linear Control of a Parallel Robot Including Motor Dynamics.* 11th RoManSy, Udine, 1-4 Juillet 1996, S. 45-52.

[142] Tönshoff, H. K.: *Vergleichende Betrachtung paralleler und hybrider Strukturen.* VDI - Berichte 1427, 1998, S. 249-270.

[143] Amirat, M. Y.; Pontnau, J.; Artigue, F.: *Force-Feedback Control of a Six DOF Parallel Robot. Application to Assembly in Car Manufacturing.* Revue d'Automatique et de Appliqée, 4(2), 1991, S. 109-121.

[144] Merlet, J.-P.: *Force - Feedback Control of Parallel Manipulators.* IEEE Int. Conf. On Robotics and Automation, Philadelphia, USA, 24.-29. April, 1988, S. 1484-1489.

[145] Hesselbach, J.; Kerle, H.; Plitea, N.: *On Some Aspects of Parallel Robots Control.* Proc. 27th Intern. Symp. On Industrial Robots (ISIR), Mailand, Italien, 1996, S. 683-687.

[146] Hesselbach, J.; Kusiek, A.: *Steuerung eines Parallelroboters für die Mikromontage.* VDI - Berichte Nr. 1427, 1998, S. 127-144.

[147] Schlittenhelm, W, u.a.: *Offen und flexibel - Konfigurierbares Bediensystem für offene Steuerungen auf PC-Basis.* Elektronik (1997), H.16, S. 52-56.

[148] Schraft, R. D.; Kaun, R.: *Stand der Technik, Defizite und Trends in der Automatisierungstechnik.* Fraunhofer Institut für Produktionstechnik und Automatisierung (IPA), Stuttgart, 1999.

[149] Kreidler, V.: *Anwendung und Leistungsfähigkeit der Steuerung 840D in Parallelstrukturen.* Tagungsband zum Chemnitzer Parallelstruktur - Seminar 28./29. April 1998, S. 205-223.

[150] Pfeifer; Thrum: *Informationstechnische Integration von Sensoren und Messsystemen in neue Steuerungskonzepte für Werkzeugmaschinen.* VDI Bericht Nr. 1255, 1996.

[151] Object Management Group: *Unified Modeling Language Specification.* Version 1.4, 2001.

[152] Object Management Group: *UML Profile for Schedulability, Performance, and Time.* Apr. 2003.

[153] Verdickt, T.; Dhoedt, B.; Gielen, F.; Demeester, P.: *Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models.* IEEE Transactions on Software Engineering, VOL. 31, NO. 8, AUGUST 2005

[154] I-Logix Inc., Products page, http://www.ilogix.com/fs_prod.htm

[155] Sztipanovits, J.; Karsai, G.: *Model-Integrated Computing.* Computer, vol. 30, no. 4, pp. 110-112, Apr. 1997.